

AD-A196 981 ROCHESTER CONNECTIONIST SIMULATOR VOLUME 1 USER MANUAL 1/2

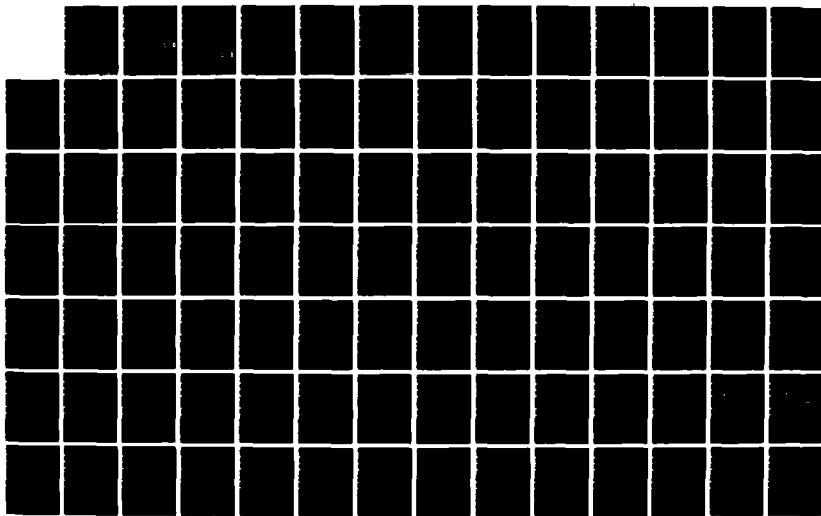
(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE

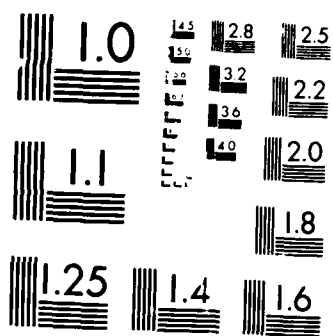
N H GODDARD ET AL 25 APR 87 TR-233-VOL-1

UNCLASSIFIED N00014-84-K-0655

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY

AD-A196 981

Rochester Connectionist
Simulator

N.H. Goddard, K.J. Lynne and T. Mintz
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 233
Spring 1987

DTIC
ELECTE
AUG 01 1988
S H D

Rochester

Department of Computer Science
University of Rochester
Rochester, New York 14627

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

88 7 29 03

Rochester Connectionist Simulator

N.H. Goddard, K.J. Lynne and T. Mintz
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 233
Spring 1987

Abstract

The Rochester Connectionist Simulator is a connectionist network simulator written in the "C" programming language and designed to be run on the UNIX operating system. This technical report comprises the User and Programmer's Manuals for the simulator. The documentation is divided into five volumes:

- (1) User Manual;
- (2) Graphics Interface User Manual;
- (3) Advanced Programming Manual;
- (4) Graphics Interface Programmer's Manual;
- (5) Back Propagation Library User Manual.

DTIC
ELECTE
S **D**
AUG 01 1988
H

This work was supported in part by the Office of Naval Research under Contract N00014-84-K-0655.

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

A116 76

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 233	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Rochester Connectionist Simulator		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) N.H. Goddard, K.J. Lynne and T. Mintz		8. CONTRACT OR GRANT NUMBER(s) N00014-84-K-0655
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Computer Science 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA / 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE Spring 1987
		13. NUMBER OF PAGES 136
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this document is unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) connectionist tools network construction neural simulator → This		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The Rochester Connectionist Simulator is a connectionist network simulator written in the C programming language and designed to be run on the UNIX operating system. This report comprises the User and Programmer's Manuals for the simulator. The documentation includes: (1) User Manual; (2) Graphics Interface User Manual; (3) Advanced Programming Manual; (4) Graphics Interface Programmer's Manual; and (5) Back Propagation Library User Manual. (KR) ←		

The Rochester Connectionist Simulator
Volume 1:
User Manual

Nigel Goddard
Dept. of Computer Science
University of Rochester
Rochester, NY 14627

April 25 1987

Contents

1 Introduction	4
1.1 Preliminaries	4
1.2 Graphics Interface	4
2 Network Construction	5
2.1 Creating space for units	5
2.2 Making units	5
2.3 Adding sites	6
2.4 Making links	6
2.5 Naming units	7
2.6 State names	7
2.7 A simple example	8
2.8 Flags	8
2.9 Sets	9
2.10 Activation Functions	10
2.10.1 Link functions	10
2.10.2 Site functions	10
2.10.3 Unit functions	11
2.11 Library functions	11
2.11.1 Unit functions	11
2.11.2 Site functions	11
2.11.3 Link functions	11
2.12 Another simple example	12
2.13 Modular construction	13
3 Making an executable Simulator	14
4 Simulation	15
4.1 Help	15
4.2 Building the network	15
4.3 Debugging during network construction	15
4.3.1 Debug command	16
4.3.2 Debug interface	16
4.3.3 Set command	17
4.3.4 Ignore command	17
4.3.5 Interrupt interface	17
4.4 Calling functions	18
4.5 Unit specification in commands	19



Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	23

per Dent

4.6	Simulation commands	20
4.6.1	Sync command	20
4.6.2	Fsync command	20
4.6.3	Async command	20
4.6.4	Go command	21
4.6.5	Echo command	21
4.7	Examination commands	22
4.7.1	Display command	22
4.7.2	List command	22
4.7.3	Show command	23
4.7.4	Pipe command	24
4.7.5	Pause command	24
4.7.6	Status command	24
4.8	Modification commands	25
4.8.1	AllocateUnits command	25
4.8.2	MakeUnit command	25
4.8.3	AddSite command	25
4.8.4	MakeLink command	26
4.8.5	NameUnit command	26
4.8.6	Out command	26
4.8.7	Pot command	27
4.8.8	State command	27
4.8.9	Weight command	27
4.8.10	Reset command	27
4.9	Set commands	28
4.9.1	Addset command	28
4.9.2	Remset command	28
4.9.3	Deleteset command	28
4.9.4	Unionset command	28
4.9.5	Intersectset command	28
4.9.6	Diffset command	29
4.9.7	Inverseset command	29
4.10	File commands	30
4.10.1	Checkpoint command	30
4.10.2	Restore command	30
4.10.3	Save command	31
4.10.4	Load command	31
4.10.5	Log command	32
4.10.6	Read command	32

4.11	Miscellaneous commands	33
4.11.1	Quit command	33
4.11.2	Print command	33
4.11.3	Printpause command	33
4.12	Abbreviated commands	34
4.13	Constructing a network from the command interface	34
5	Advanced Programming Features	35
5.1	Simulator Variables	35
5.2	Network Access Functions	36
5.2.1	Display functions	36
5.2.2	Naming	36
5.2.3	Simulating	36
5.2.4	Modifying and Accessing the Network	37
5.2.5	Unit macros	37
5.2.6	Miscellaneous library functions	37
5.3	Saving and reloading user data structures	38
5.4	Customizing unit, site and link data structures	38
5.5	Customizing the simulator command interface	38
5.6	Modelling Time	38
6	An Extended Example of network construction	39
6.1	Designing a network	39
6.2	The Problem: four coloring a map	39
6.3	Designing the build program	40
6.4	Implementing in C	41
6.4.1	Top level <i>build</i> function	41
6.4.2	Making a region	43
6.4.3	Making borders	45
6.4.4	The unit function	46
6.5	A command script to demonstrate the network	47
7	Floating Point version	48
8	Acknowledgements	48

1 Introduction

Connectionist networks consist of simple computational elements (units) which communicate by sending their level of activation via links to other elements. The units have a small number of states, and compute simple functions of their inputs. Associated with each link is a weight, indicating the "significance" of activation arriving over that link. The behaviour of the network is determined by the pattern of connections, the weights on the links, and the unit functions.

The Rochester Connectionist Simulator supports construction and simulation of a wide variety of networks. The main design criterion has been flexibility. Each unit can compute a different function, any amount of data may be associated with each unit, and an arbitrary connection pattern may be specified.

The particular network paradigm supported by the simulator is, in brief, as follows. Each unit has a number of *sites* at which the incoming links arrive. The provision of sites allows differential treatment of inputs, since the links themselves do not indicate their origin at the destination unit.

Simulations may be run synchronously or asynchronously. During synchronous simulation all units use the output values computed during the previous step as their input. The order of simulation is unimportant, the network behaving as though all units update simultaneously. During asynchronous simulation, at each step a fraction of the units are updated, in pseudo-random order, with the new output value immediately transmitted to the other units. It is guaranteed that after a limiting number of steps every unit will have updated at least once.

There are a number of example networks in the `/example` directory. It would be a good idea to check these out before writing any code.

1.1 Preliminaries

It will aid in understanding the following sections if the representation of units, sites and links is discussed here. The main data structure is an array of unit structures (of type `Unit`), with a linked list of site structures (of type `Site`) attached to each unit structure, and a linked list of incoming link structures (of type `Link`) attached to each site structure. Unit, site and link structures contain various pieces of data. Units have a *potential*, corresponding to the level of activation, a *state* which can be used to vary the unit function, and an *output* which is transmitted along the outgoing links. Each site has a *value*, which is set by the site function. Each link has a *weight* which may be modified by the link function, a pointer *value* to the incoming value, and the index *from_unit* holding the index of the source unit for the link. Each unit, site and link structure contains a pointer to a function which is called by the simulator to simulate the action of the unit, site or link. In addition, each unit, site and link structure contains a *data* field which is for general use by user. Further details may be found in the Advanced Programming Manual.

1.2 Graphics Interface

The simulator command interface (see section 4) was designed for simple terminal operation. This document describes network construction and simulation on the terminal interface.

The *Graphics Interface* is a package that runs on top of the simulator described in this document, and is a powerful network debugging aid. See the document *The Rochester Connectionist Simulator: Graphics Interface User Manual* for details. If the Graphics Interface is included when making a simulator (see section 3), it is automatically invoked when the simulator is run. All the commands described in section 4 are available from the Graphics Interface, as well as many more.

2 Network Construction

Although it is possible to construct a network from the command interface (see section 4.13, it is a time consuming process and really only suited to novice users. One of the example networks is constructed this way, and is described in section 4.13.

Generally a network is built in the simulator by a user program written in C. The simulator provides primitive functions, the major ones being to make units, add sites, make links, and associate a name with one or more units. Many other primitives are also provided to access parts of the network data structure.

2.1 Creating space for units

Before any units can be made, the program should specify the total number of units needed. The program may only ask for units once, but need not actually use all the units asked for. The total number of units is specified with a call to `AllocateUnits`, for example:

```
AllocateUnits(100);
```

This allocates data space for the requested number of units. If a program does not explicitly allocate space for units, then by default space for 200 will be allocated.

2.2 Making units

Now units may be made with a call to `MakeUnit`. This function builds a new unit, using space allocated by `AllocateUnits`, for example:

```
int MakeUnit(type,func,init-pot,potential,data,output,init-state,state)
    char *type;
    func_ptr func;
    int istate, state, init-pot, potential, output, data;
```

type is a pointer to a character string, and is simply used for display purposes. *func* is a pointer to the function used to simulate the unit's action. *potential* is the activation level for the unit. *data* is a four byte value for the unit *data* field described above. *output* is the initial output of the unit. *state* is a short integer representing the initial state value. *init-pot* and *init-state* are the values to set the unit potential and state when the network is *reset*. `MakeUnit` returns the index in the unit array of the unit created. The first call to `MakeUnit` builds the unit with index 0, and consecutive calls to `MakeUnit` will return consecutive indices. An example of a call to `MakeUnit` would be:

```
unit-index = MakeUnit("retinal",UFsum,500.500,0.50,1.1);
```

The function pointer may be `NULL`, in which case a function which does nothing will be called by the simulator to simulate the unit action. If not `NULL`, the function must be either one you have written, or one of the library functions. For simple networks the library functions (see section 2.11) should be sufficient.

2.3 Adding sites

Once a unit has been created, one or more sites may be attached to it with calls to AddSite:

```
Site * AddSite(index, name, function, data)
    int unit, data;
    char *name;
    func_ptr func;
```

index is the index of the unit to which the site is to be attached. *name* is a pointer to a character string which will be the name of the site. *function* is a pointer to the function to be called to simulate the action of the site. *data* is the four byte value to be placed in the site *data* field described above.

Links to the unit cannot be made until there is a site attached to the unit to which they may go. A call to AddSite might look like:

```
AddSite(unit-index, "excite", SFweightedsum, 0);
```

AddSite returns a pointer to the newly created site structure. As with units, the function may be NULL, one of your functions, or one of the library functions.

2.4 Making links

A link from a unit to a site on another unit is created with a call to MakeLink.

```
Link * MakeLink(source-unit, destination-unit, site-name, weight, data, function)
    int from, to;
    int weight, data;
    char *site;
    func_ptr func;
```

source-unit is the index of the unit where the link originates. *destination-unit* is the index of the unit to which the link is going. *site-name* is a pointer to a character string which is the name of the site on the destination unit at which the link is to arrive. *weight* is the weight to put on the link, and should be within range of a short integer. By convention weights are scaled down by a factor of 1000, thus a specified weight of 500 will be treated as a weight of 0.5. This is to allow weights in the range 0 to 1 without having to use floating point arithmetic. Weights may be negative. MakeLink returns a pointer to the link structure created. An example of a call to MakeLink might be:

```
MakeLink(unit-index, unit-index, "excite", -500, 0, LFsimple);
```

This would make a link from the unit to itself, to be attached at the site "excite", with a weight of -500 (meaning -0.5), and function LFsimple. Such a link could be used to provide exponential decay. As with units, the function may be NULL, one of your functions, or one of the library functions.

2.5 Naming units

Each unit may be given a name with a call to NameUnit:

```
NameUnit(name,type,index,length,depth)
char *name;
int type,index,length,depth;
```

As well as naming a single unit, this function can name a vector or 2-D array of units. The name may then be used during simulation from the command interface (see section 4), and may also be used during network construction. *name* is a pointer to the character string name to be given. *type* is the type of name: SCALAR, VECTOR, or ARRAY. *index* is the index of the unit to be named, or the first unit in the vector or array. *length* is the number of units if it is a VECTOR, and the number of columns if it is an ARRAY, and is undefined for SCALAR. *depth* is the number of rows for an ARRAY, and is undefined for SCALAR and VECTOR.

A name which is specified as VECTOR or ARRAY will apply that name to the unit with the index specified, and to the requisite number of units following it in the unit array. Thus the call

```
NameUnit("Vertex", ARRAY, 100, 4, 2);
```

will apply the name "Vertex" to units 100 through 107, making an array of 2 rows of 4 units. Now, for example, unit 107 will be displayed by the simulator with the name "Vertex[1][3]".

All names must be unique. This applies to state, site, function, type and set names, as well as unit names.

2.6 State names

A name may be given to a state. Internally the state is just a short integer, but for display purposes it is much clearer if the state of a unit is printed as a name rather than merely a number. A state name is declared with, for example:

```
DeclareState("active",1);
```

which assigns the name "active" to the state represented by number 1. Now whenever a unit is displayed, if its state field has value 1 the simulator will print the name "active", otherwise it will print the state value as an integer. A maximum of 100 state names may be declared, corresponding to state values 0 to 99.

2.7 A simple example

The following sample program will build a network of 10 units, each one linked to all the others with inhibitory links. This kind of structure is known as a winner-takes-all network.

```
build()
{
    int i, j, index;

    AllocateUnits(10);
    for (i = 0; i < 10; i++)
    {
        index = MakeUnit("competing", UFsum, 0, 0, 0, 0, 1, 1);
        AddSite(index, "inhibit", SFweightedsum, 0);
    }
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
            if (i != j)
                MakeLink(i, j, "inhibit", 0-(random()%1000), 0, NULL);
    DeclareState("active", 1);
    NameUnit("W-T-A", VECTOR, 0, 10, 0);
}
```

First space for the 10 units is allocated. Then, in the first for loop the 10 units of type "competing" are constructed, and a site with name "inhibit" is added to each one. The functions UFsum and SFweightedsum are library functions. UFsum simply sets the unit potential and output to be the sum of the values of all the units sites (in this case, just one). SFweighted sum sets the value of the site to be the weighted sum of the incoming link values. All the initial values for the unit fields are set to zero, apart from the states, which set to 1.

The second for loop constructs all the links: one from each of the other units, all attached to the sites "inhibit". The weights are set to a random value in the range 0 to 1000 (representing 0 to 1). The name "active" is associated with state value 1, and the 10 units are named as a vector "W-T-A". The first unit built is always unit 0, so the index in the call to NameUnit can safely be assumed to be 0.

2.8 Flags

Each unit has 32 flags associated with it. Currently flags 0 to 6 are used by the simulator, and flags 7 to 11 are reserved for future simulator use. Flags 12 to 19 should be used for library packages, and so user code should be restricted to flags 20 through 31, preferably working from 31 down. Some of the simulator reserved flags may be set by the user for one or more units.

The user-settable flags are as follows:

SHOW_FLAG	if set then the unit is in the Show set (see section 4.7.3).
LIST_FLAG	if set then the unit is in the List set (see section 4.7.2).
NO_LINK_FUNC_FLAG	if set then no functions are called for the links into a unit. This will result in speed up.
NO_SITE_FUNC_FLAG	if set then no site or link functions are called for the unit.
NO_UNIT_FUNC_FLAG	if set then no unit, site or link functions are called for the unit. The output of the unit remains the same.

Flags may be set when the network is constructed (i.e. in the build program) or during simulation (i.e. by unit functions), or in fact by any user function. The macros used to set, clear, and test flags are:

```
SetFlag(unit-index, flag)
UnsetFlag(unit-index, flag)
TestFlag(unit-index, flag)
```

TestFlag computes TRUE if the flag is set for the unit, FALSE otherwise. For efficiency purposes, macros which use pointers to the unit (to avoid indexing into the unit array) are also available: SetFlagP, UnsetFlagP, TestFlagP (unit-pointer, flag). These enable flag setting in a tight loop in an efficient manner.

2.9 Sets

The user may create, modify and delete sets of units. The maximum number of sets at any instant is 32. Unless otherwise stated, the set functions return TRUE if the function succeeded, and FALSE otherwise. In general the return value will only be FALSE if any of the sets specified do not exist, or cannot be created. The set functions are:

DeclareSet(name) - creates a set.

DeleteSet(name) - deletes a set.

AddToSet (name, low, high) - adds unit with indices low through high to the set.

RemFromSet (name, low, high) - removes units low through high from the set.

UnionSet (name3, name1, name2) - assigns the union of sets name1 and name2 to the set name3. Creates set name3 if it does not already exist.

IntersectSet (name3, name1, name2) - assigns the intersection of sets name1 and name2 to the set name3. Creates set name3 if it does not already exist.

DifferenceSet (name3, name1, name2) - assigns all units in set name1 but not in set name2 to the name3. Creates set name3 if it does not already exist.

InverseSet(name1, name2) - assigns all units not in set name2 to be in set name1. Creates set name1 if it does not already exist.

MemberSet (name, unit-index) - returns TRUE if the unit is in the set with the given name, FALSE otherwise.

IsSet(name) - returns TRUE if the name is the name of a set, FALSE otherwise.

Sets are a useful way to impose some structure on an otherwise amorphous mass of units. Unit functions may add or subtract a unit from a set. The sets are known to the command interface (see section 4) by name when simulating, and so simulation commands can be applied to them.

2.10 Activation Functions

The library contains some standard unit, site and link functions, but user-written functions may be used. Unit, site and link functions are called by the simulator during each time step.

2.10.1 Link functions

Let us look at the library function *LFsimple* for a sample link function.

```
LFsimple(up,sp,lp)
    Unit *up;
    Site *sp;
    Link *lp;

{
    lp->data = *(lp->value);
}
```

Three pointers are passed as parameters when the simulator calls a link function: a pointer to the unit to which the link is attached; a pointer to the site at which the link arrives; and a pointer to the link itself. *LFsimple* uses only the link pointer, and simply stores the incoming value in the link data field. In effect the data field is being used as a one-simulation-step memory. Check the Advanced Programming Manual for complete details of Link structure. Note that the *value* field in the Link structure is a pointer.

2.10.2 Site functions

A commonly used site function, from the library, is *SFweightedsum*:

```
SFweightedsum(up,sp)
    Unit *up;
    Site *sp;

{
    int sum;
    Link *lp;

    for(lp = sp->inputs, sum = 0; lp != NULL; lp = lp->next)
        sum += (*(lp->value) * lp->weight);
    sp->value = sum/1000;
}
```

Pointers to the unit to which the site is attached, and to the site itself, are passed in as parameters to site functions. *SFweightedsum* simply trips down the linked list of incoming links, accumulating the weighted sum of the incoming values. The end of the list is terminated with a NULL *next* field in the final Link structure. The weighted sum is divided by 1000 (the weight scaling factor), and the result set in the site *value* field.

2.10.3 Unit functions

One of the simplest possible unit functions is *UFsum*, which simply sums the site values:

```
UFsum(up)
    Unit *up;

{
    int sum;
    Site *sp;

    for(sp = up->sites, sum = 0; sp != NULL; sp = sp->next)
        sum += sp->value;
    up->output = up->potential = sum;
}
```

Unit functions are passed a pointer to the unit structure when called by the simulator. This function trips down the linked list of sites attached to the unit, summing up the values. The final sum is set in the *potential* and *output* fields of the unit structure, thus setting the value that will be transmitted along outgoing links.

2.11 Library functions

2.11.1 Unit functions

UFsum is a unit function which sets output and potential to the sum of all site values.

2.11.2 Site functions

SFmax sets the site value to the maximum input value.

SFmin sets the site value to the minimum input value.

SFsum sets the site value to the sum of the input values.

SFweightedmax sets the site value to the maximum weighted input value. A weight of 1000 is treated as unity: the input value is multiplied by its weight and the result divided by 1000.

SFweightedmin(up.sp) sets the site value to the minimum weighted input value. A weight of 1000 is treated as unity.

SFweightedsum sets the site value to the sum of the weighted input values. A weight of 1000 is treated as unity.

SFand returns 1 if all its inputs are positive, otherwise 0.

SFxor(up.sp) returns 1 if exactly one of its inputs is nonzero, otherwise 0.

SFprod returns product of inputs.

2.11.3 Link functions

LFsimple sets the data field of the link to be the input value (unweighted). This does not affect the behavior of the network, but does help with debugging.

2.12 Another simple example

Let us modify the winner-takes-all example to demonstrate the use of sets and flags

Since the link functions were specified to be NULL, i.e. a function which does nothing, we could equally well set the NO_LINK_FUNC_FLAG for each unit, saving the time taken to call the null function for each link.

We shall also create a set "still-competing" which will contain all the units whose *potential* is greater than 0. This set could then be displayed during simulation to view the winner-takes-all inhibition process.

build()

```
{
    int i, j, index;

    AllocateUnits(10);
    for (i = 0; i < 10; i++)
    {
        index = MakeUnit("competing".UFmysum, 0, 0, i, 0, 1, 1);
        AddSite(index, "inhibit", SFweightedsum, 0);
        SetFlag(i, NO_LINK_FUNC_FLAG);
    }
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
            if (i != j)
                MakeLink(i, j, "inhibit", 0-(random()%1000), 0, NULL);
    DeclareState("active", 1);
    NameUnit("W-T-A", VECTOR, 0, 10, 0);
    DeclareSet("still-competing");
    AddToSet("still-competing", 0, 9);
}
```

The network building program has changed very little. The NO_LINK_FUNC_FLAG is set for each unit as it is made, and the unit's *data* value is set to be the index of the unit. At the end of the program we declare the set "still-competing" and add the 10 units to it. But the major change is we no longer use the library function *UFsum*. Instead we write our own function, *UFmysum*, which in addition to setting the *output* and *potential*, adds or removes the unit from the set.

```
UFmysum(up)
    Unit *up;

{
    int sum;
    Site *sp;

    for(sp = up->sites, sum = 0; sp != NULL; sp = sp->next)
        sum += sp->value;
    up->output = up->potential = sum;
    if (sum <= 0)
        RemFromSet("still-competing", up->data, up->data);
}
```

As each unit is simulated, it checks if it is still in the competition; if not, it removes itself from the "still-competing" set, using its index which was stored in the *data* field when the unit was created.

Now if the set "still-competing" is displayed at every step during simulation, the winner-takes-all process will become apparent.

2.13 Modular construction

One of the most important aspects of network construction is a modular approach. The actual size and configuration of network can be specified at the highest level in a data file, containing an abstract version the problem being modelled. At the next level, the build function is used to control the gross aspects of network construction. Another level down, separate functions can be written to construct the different types of units and links. This corresponds to a *hierarchy of descriptive levels* and is crucial for building large networks. Section 6 gives an example of this approach.

3 Making an executable Simulator

Before a network can be simulated, the program to build it must be compiled and linked in with the simulator object files. A shell script makes this task simple. The name of the shell script is *makesim*, and it is normally found in the *~connect/bin* directory - but this is site dependent. There are a number of flags which may be specified, if you specify none it will assume you are running on a SUN workstation and will create an integer simulator with the graphics interface. To create such a simulator, assuming the network building program is in the file *build.c*, simply type:

```
makesim build.c
```

This will create an executable binary file *sim* which is the simulator. When *sim* is run, the graphics interface window will appear and simulation commands, described in the next section, may be executed.

The complete specification for *makesim* is given in the man page. The simulator man directory, *~connect/man*, should be on the manpath. The most commonly used flags are:

-ng	do not load graphics interface.
-T	assume not running on a SUN.
-g	compile user code for debugging.
-r	run the executable when it has been created.
-d	run the executable under dbx(tool) when it is created.
-o <file>	file name for the executable.

Multiple C files ending with *.c* and object files ending with *.o* are allowed. For instance, the line:

```
makesim build.c initialize.c test.o print.o
```

would compile the files *build.c* and *initialize.c* and load the resulting object files with *test.o* and *print.o* and the simulator object files, to make the executable.

Libraries can also be included. The line:

```
makesim build.c mylib.a neurolib.a
```

would compile the file *build.c* and load the resulting object file with the simulator object file, together with appropriate code extracted from the library files *mylib.a* and *neurolib.a*.

4 Simulation

When a simulator executable is run, the startup message will be printed and commands can be typed when the prompt appears. The startup message will be something like:

```
Rochester Connectionist Simulator 4.0

integer version

Debugging turned on, not in Auto-Fix mode
->
```

The command interface prompt is `->`. The startup message indicates what kind of simulator is running, integer or floating point. Debugging and Auto-Fix are described below. The simplest command is `?`. Typing this to the prompt will result in all the command names being listed.

4.1 Help

The *help* command has the syntax:

```
help [<item>]
<item> ?
```

help alone will print the standard help message, describing the types of commands available, and how to exit the simulator. *help name* and *name ?* are equivalent, and print help information about *name*. This should be a command name or the special item *UnitId*. For instance, the command *help display* will print information as to what the *display* command does. Many of the commands require a specification of a unit or range of units, which can involve unit indices, unit names, set names or a combination of these. *help UnitId* will print information as to how to specify units for commands.

4.2 Building the network

The network building program was linked in with the simulator to produce the executable which is now running. Before the network can be simulated, it must be constructed by running this network building program. By convention, the top level function in the network building program is *build*. To execute this function, the *call* command (see section 4.4) is used, in this case:

```
call build
```

4.3 Debugging during network construction

It is often the case that network building programs contain errors, for instance specifying a link between two units, one of which does not exist. The simulator can be put in *debug* mode, in which case these kinds of errors do not cause a core dump, but rather are automatically fixed, or cause a *debug interface* to be entered.

4.3.1 Debug command

Syntax: `debug [[auto] <on | off>]`

Example: `debug auto on`

The debug command is used to switch network construction debugging on and off, and to switch automatic error correction on and off. *debug on* and *debug off* will control whether debugging is operative. *debug auto on* and *debug auto off* will control whether automatic error correction is in operation. If automatic error correction is switched on and a log file is being maintained, then the user may accumulate a list of errors to be fixed, just as a conventional compiler produces a list of errors before aborting.

Debugging only applies to network construction, not to simulation. If debugging is switched on, then anytime the simulator tries to make a unit, site or link, the simulator will check that the values specified are suitable. For instance, if a function for a unit that is not known to the simulator is specified, the simulator will issue an error message. If automatic correction (Auto-Fix) is switched on, the simulator will substitute what it thinks is a suitable value, and continues.

If Auto-Fix is off, the simulator will enter a debug interface (see section 4.3.2) and asked for the errors to be fixed. The *set* command (see section 4.3.3) is used to change incorrect values. Once all the errors have been fixed, the *quit* command may be used to return to the normal network construction process. Continuation is not possible until the errors have been fixed, unless the *ignore* command (see section 4.3.4) is issued.

If debugging is turned off, absolutely NO CHECKING is done - a core dump will occur if anything is wrong during construction. It is best to build and test a network with debug switched on until it is clear that the build function is correct. Building without checking appears to be approximately twice as fast as with checking.

If debugging and Auto-Fix are switched on, and errors or warnings are occurring rapidly, typing *control.C* which will introduce the interrupt interface (see section 4.3.5) whence the network may be examined, Auto-Fix switched off, etc. If *control.C* is typed when the network is not in a safe state, the interrupt will be delayed.

4.3.2 Debug interface

The debug interface is signalled by a different prompt:

```
debug[n]>
```

where *n* is an integer indicating the interface level. Level 0 is the normal command interface. Most of the commands available at level 0 are available to the debug interface, but none of the commands that cause a simulation step, such as *go* or *read*. When the debug interface is entered, a list of errors that need fixing, and for which unit, site or link, is displayed. The list of errors may be displayed at any time with the *set* command. When all the errors have been fixed with the *set* command, exiting the debug interface with *quit* will cause the building process to continue. Exiting with *ignore* will cause the building process to continue, but the unit, site or link will not have been made.

The way in which debug levels may accumulate, is that on being put in the debug interface, say level 1, the user may decide that a unit should be made on the fly, with the *MakeUnit* command. If the specification given for this unit is incorrect, the next level debug interface will be entered to fix this specification. On exit from a debug interface, if one returns to another lower level debug interface then the *set* command may be used to recall what the errors were that caused this interface to be entered originally.

4.3.3 Set command

Syntax: set <pot|out|state|ipot|istate|unit|to|from|weight> <value>
 set <func|type|site> <name>
 set all default
 set

Example: set func SFweightedsum

The *set* command, which is only available at the debug interface, is used to correct errors in unit, site or link specifications. Errors may be fixed with specific values, or the simulator default values may be used: *set all default*. The list of outstanding errors will be printed in response to: *set*. It will also be printed if the user attempts to *quit* before all the errors have been corrected. For example, suppose one tried to build a unit with the *MakeUnit* command (see section 4.8.2):

```
-> MakeUnit mytype NullFunc 1 2 3 4 1234567 1234567
The following errors were encountered while trying to
make unit 1 of type mytype and function NullFunc.
- initial state value 1234567 out of range -/+32767
- state value 1234567 out of range -/+32767
debug[2]> quit
There are still errors in the unit definition:
- initial state value 1234567 out of range -/+32767
- state value 1234567 out of range -/+32767
debug[2]> set
Current values for MakeUnit are:
- type = mytype
- function = NullFunc
- state = 1234567
- init state = 1234567
- potential = 2
- init potential = 1
- output = 4
- data = 3
remaining errors are:
- initial state value 1234567 out of range -/+32767
- state value 1234567 out of range -/+32767
fix and quit, or type ignore
debug[2]> set all default
debug[2]> quit
Made unit 1
->
```

4.3.4 Ignore command

Syntax: ignore

The *ignore* command, which is only available at the debug interface, is used to continue the construction process without constructing the unit, site or link whose specification was in error. This may be expedient, but may also cause further errors in the construction process at a later time.

4.3.5 Interrupt interface

The interrupt interface is entered whenever the user types control.C. The interrupt interface is signalled by a different prompt:

`interrupt[n]>`

where *n* is an integer indicating the interface *level*. Level 0 is the normal command interface. Most of the commands available at level 0 are available to the interrupt interface, but none of the commands that cause a simulation step, such as *go* or *read*. If `control.C` is typed during a piece of guarded code, a message will be printed and entry to the interrupt interface will be delayed until the guarded code is exited.

The intention behind the interrupt interface is that the user can first of all manipulate the debug settings (whether on or off, whether automatic or manual fixing), and secondly can interrupt a potentially catastrophic situation to save or examine the network before it is corrupted or destroyed.

4.4 Calling functions

Any function in the user code can be called from the interface, using the *call* command, so long as the function was not declared static in the code (see the C manual for an explanation of this). The *call* command has the syntax:

Syntax `call function-name [<arg1> <arg2>]`

Example `call Initialize image.inp.1`

This will call the function with the optional arguments as parameters. The only required use of this command is to construct the network, as described in the preceding section. An example, as above, of an additional use would be to call a function which initializes a set of units representing a retinal image array from file data.

4.5 Unit specification in commands

Many commands expect one or a range of units to be given, to which the command will be applied. A single unit may be specified by index or by name. A range of units may be specified by the low and high units separated by - (with mandatory space either side of the -), by a set name, by a VECTOR or ARRAY name, or by the token *all*.

For example, suppose units 0 to 9 have names ZERO, ONE, TWO, ..., NINE, units 10 to 19 are a vector with name MyVector, units 20 to 39 are a 4 by 5 array of units with name MyArray, set FirstSet consists of all the odd-index units, and these 40 units are all that have been made.

Then the following unit specifications are valid and indicate these units:

Specification	Units
0	0
ONE	1
0 - 3	0.1.2.3
THREE - 5	3.4.5
MyVector	10.11.12.....19
MyArray	20.21.22.....39
MyVector[4] - 19	14.15.16.17.18.19
MyArray[1][4] - MyArray[3][4]	29.30.31.....39
FirstSet	1.3.5.....39
all	0.1.2.....39

As can be seen, mixed modes are allowed.

4.6 Simulation commands

The simulation commands are used to modify aspects of the simulation, and to cause one or more steps to be simulated.

4.6.1 Sync command

Syntax: `sync`

The `sync` command sets the simulation to be synchronous. This means that at each simulation step, every unit will be updated, using the output values calculated from the previous step. The network behaves as if all units update simultaneously. This is the default setting for the update protocol.

4.6.2 Fsync command

Syntax: `fsync <execfrac> <execlimit> [<random seed>]`

Example: `fsync 10 100 2039`

The `fsync` command sets the simulation style to be asynchronous. This means that at each time step a percentage of units (given by `execfrac`) picked pseudo-randomly from all the units are simulated, and that after a limiting number of steps (given by `execlimit`) all units will have been simulated at least once. The new output of a unit is available immediately the unit is simulated, for other units to use. Thus the network should not be sensitive to the order in which units are simulated. The point by which all units must have been simulated at least once (`execlimit`) is determined with reference to the simulator Clock (see section 5.1). When the clock is an exact multiple of `execlimit`, any unit that has not been simulated is simulated.

If an integer number is given for `random seed` then the random number generator will be seeded with this number. Seeding with the same number on separate occasions will cause the generator to produce the same sequence of random numbers, thus by specifying the same seed a session involving asynchronous simulation can be repeated exactly. If no seed is specified, the UNIX system time will be used.

This style of execution becomes more inefficient as the execution fraction increases (for implementation reasons). For an execution fraction of 100%, the `async` command should be used.

4.6.3 Async command

Syntax: `async [<random seed>]`

Example: `async 2039`

This command is a special case of the `fsync` command. Logically it has the same effect as if `fsync 100 1` were given, i.e. that at each time step all units will be simulated in pseudo-random order. For implementation reasons it much better to use this command than the equivalent `fsync` command.

4.6.4 Go command

Syntax: `go [clock] [<StepCount>]`

Example: `go clock 100`

The *go* command causes one or more simulation steps to be run. *StepCount* specifies the number of steps to run, and defaults to 1. If the *clock* option is used, the simulator will time how long it takes to simulate the number of steps, in increments of 1 second. This can be used to get a precise idea of network code efficiency. Since the time quantum is 1 second, a significant number of steps must be simulated to get reliable timing data.

4.6.5 Echo command

Syntax: `echo [StepCount | on | off]`

Example: `echo 10`

The *echo* command sets how often the simulator prints the echo message:

finished *x* out of *n* steps

Echoing occurs (if it is switched on) during execution of a *go* command. *n* is the number of steps specified in the *go* command, and *x* is a multiple of the *StepCount* specified in the *echo* command. For example:

```
-> echo 5
-> go 20
finished 5 out of 20 steps
finished 10 out of 20 steps
finished 15 out of 20 steps
finished 20 out of 20 steps
->
```

Echoing is switched off with: *echo off*; and on with: *echo on*. At start up echo is switched on. Issuing the *echo* command without any of the options will cause the current setting to be displayed.

4.7 Examination commands

The examination commands are used to print out details of units and links, either automatically or at the user's request.

4.7.1 Display command

Syntax: `disp unit <UnitID>`

Example: `disp unit 33`

The *disp* command is used to display the values associated with one or more units, for instance the potential, output, state, functions, site names and values, link weights and values. The name and index of the unit where the link originates is shown in each link display. For example:

```
-> disp unit 0
Unit:0 Name:W-T-A[0] Type:competing function:UFsum
potential:10 output:10 state:active data:0
Set memberships: still-competing
sitename:inhibit function SFweightedsum value:0 data:0
  link:W-T-[7] (7) func:NullFunc weight:873 value:0 data:0
  link:W-T-A[4] (4) func:NullFunc weight:216 value:0 data:0
  link:W-T-A[3] (3) func:NullFunc weight:477 value:0 data:0
->
```

4.7.2 List command

Syntax: `list <link | set | unit UnitId>`

Example: `list unit 3 - 5`

The *list* command is used to display information about links, sets or units. *list link* displays all the links. *list set* displays all the set names and the number of remaining slots for sets. *list unit UnitId* will display in compact form the units specified by *UnitId*. For example:

```
-> list unit 3 - 5
```

```
Clock = 0
```

Index	Name	Type	Potential	Output	State
3	••NO NAME••	vertex	0	0	0
4	••NO NAME••	vertex	0	0	0
5	••NO NAME••	vertex	0	0	0

```
->
```

4.7.3 Show command

Syntax: show [on | off]
 show <step | pot > <value>
 show +/- <UnitId>
 show set [(+|-) <set name>]
Example: show set + still-competing
 show + 3 - 5

The *show* command is used to control what information is displayed during simulation. Showing is turned on or off with *show on* or *show off*. If showing is turned on, then every *n* steps the set of units selected for showing (the Show set) is displayed in compact form, where *n* is set with *show step n*.

The Show set is controlled in various ways. If a unit has a potential greater than or equal to the Show potential, it is in the show set. The Show potential is set with *show pot value*, and is initially a very large number. A unit or range of units may be added to the Show set with *show + UnitId*, and removed from the Show set with *show - UnitId*. A set of units may be added to or removed from the Show set with *show set +/- set-name*. Although *show + set-name* is equivalent to *show set + set-name*, the latter is more efficient. For example:

```
-> show + 3 - 5
-> show on
-> show step 2
-> go 2
```

```
finished 1 out of 2 steps
finished 2 out of 2 steps
```

Index	Name	Type	Potential	Output	State
3	**NO NAME**	vertex	0	-16643	0
4	**NO NAME**	vertex	0	-31767	0
5	**NO NAME**	vertex	0	-24072	0

```
->
```

4.7.4 Pipe command

Syntax: pipe < on | off | command >

Example: pipe /usr/ucb/more

Output from *display*, *list*, and *show* commands can be fed into a pipe rather than directly to the screen. The default pipe is the UNIX *more* command: this simply avoids large displays scrolling off the screen. Another use might be to save displays to a file, with for example:

```
pipe cat >> save.file
```

If piping is turned on (*pipe on*) then at every *display*, *list* or *show*, the output is sent to the pipe. Piping may be turned off with *pipe off*.

4.7.5 Pause command

Syntax: pause < on | off >

Example: pause on

The *pause* command is used to avoid displays scrolling off the screen. If pausing is switched on, then after every *show* the simulator waits for user indication to continue.

4.7.6 Status command

Syntax: status

The *status* command displays the state of the simulator. For example:

```
-> status
Clock: 5           Show is on
NoUnits: 10        ShowPot: 3
NoLinks: 30        NoSets: 0
Echo every 1 steps Pause is on
Pipe is on         PipeCommand is /usr/ucb/more
Simulation is synchronous
->
```

4.8 Modification commands

There are two types of modification commands: those that alter the network *structure*; and those that alter the *state* of the network. The former commands are those to make a unit, site, or link. In addition the commands to allocate data space for units and to give a name to one or more units are included in this section.

4.8.1 AllocateUnits command

Syntax: AllocateUnits <number>

Example: AllocateUnits 200

AllocateUnits is used to create data space for the units. It should be called before any call to MakeUnit, and can only be called once. This means you should allocate more than enough units for your purposes all at once.

4.8.2 MakeUnit command

Syntax: MakeUnit <type> <function> <ipot> <pot> <data> <out> <istate> <state>

Example: MakeUnit competitor Ufsum 0 0 0 0 1 1

MakeUnit is the simulator command to make one unit. It takes the same arguments as the simulator function to make a unit (see section 2.2), namely type, function, initial potential, potential, data, output, initial state, and state, in that order. The first parameter, *type*, is simply stored in the name table by the simulator for display purposes. Any name that is not already in use (i.e. not a name of a set, state, function or unit(s)), may be used here. The second parameter, *function*, is the unit function, which must be known to the simulator (i.e. a library function or a function in user files, or NULL for the function which does nothing). The remaining arguments are integer values for the various fields of the unit. Initial potential is the unit potential after a reset. Potential is the current unit potential. Data is for you to use as you wish. Output is the current unit output. Initial state is the unit state after a reset. State is the current unit state.

Defaults for the numeric arguments are zero. Default for the function is the function NullFunc (does nothing). Default for type is the name NullType. Debugging is automatically switched on for the duration of this command.

There must be space for the unit to be made. If a build program has been run, and unit space already allocated, then the unit will be made as long as enough space remains. If no unit space has yet been allocated, i.e. *AllocateUnits* has not been called, then space for 200 units will automatically be allocated when you issue the *MakeUnit* command.

4.8.3 AddSite command

Syntax: AddSite <unit> <sitename> <function> <data>

Example: AddSite 9 "excite" SFweightedsum 0

AddSite is the simulator command to add a site to a unit. It takes the same arguments as the simulator function to add a site (see section 2.3), namely: unit index, site name, site function, and site data. A site with the given name and function is attached to the unit with the given index, and the data field is set as given. Any name that is not already in use (i.e. not a name of a set, state, function, type or unit(s)) may be used. The site function must be known to the simulator (i.e. a

library function or a function in user code or NULL for the function that does nothing). The unit index must be that of an existing unit. Data is for general use.

Defaults exist for site name, site function and data are NullSite, NullFunc and 0 respectively. The unit to which site is to be attached MUST be specified. Debugging is automatically switched on for the duration of this command.

4.8.4 MakeLink command

Syntax: MakeLink <from> <to> <site> <weight> <data> <function>

Example: MakeLink 3 5 "excite" 500 0 LFsimple

MakeLink is the simulator command to make a link from one unit to another. It takes the same arguments as the simulator function to make a link (see section 2.4), namely: source unit index, target unit index, site name (on target unit), link weight, link data and link function. A link from the source unit to the named site on the target unit is made with function, weight and data as given. The link function must be known to the simulator (i.e. a library function or a function in your files, or NULL for the function which does nothing).

The specified weight is scaled down by the simulator by a factor of 1000, so 500 corresponds to a real weight of 0.5. *data* is for general use. Defaults exist for function (NullFunc), weight (0) and data (0). The source and target unit indices, and the name of the site on the target unit MUST be specified. Debugging is automatically switched on for the duration of this command.

4.8.5 NameUnit command

Syntax: NameUnit <scalar|vector|array> <index> [<width> [<depth>]]

Example: NameUnit array 10 4 5

NameUnit is used to give a name to one or more units. It takes the same arguments as the simulator function of the same name, i.e. a *name*, a *type*, the *index* of the first unit to which the *name* is to be applied, the *width* (if a vector or array), and *depth* if an array. The possible types are: scalar, vector, array. For a scalar name, the name is applied to a single unit. For a vector name, the name is applied to *width* units starting with *index*. For an array name, the name is applied to *width*depth* units starting with *index*.

4.8.6 Out command

Syntax: out <UnitID> <value> [<UnitID> <value>]*

Example: out 3 100 4 200 5 300

The *out* command is used to set the *output* of one or more units. It expects one or more unit-identifier/output-value pairs. The unit identifiers may be specified in any of the usual ways (see section 4.5).

4.8.7 Pot command

Syntax: `pot <UnitID> <value> [<UnitID> <value>]*`

Example: `pot 3 100 4 200 5 300`

The *pot* command is used to set the *potential* of one or more units. It expects one or more unit-identifier/potential-value pairs. The unit identifiers may be specified in any of the usual ways (see section 4.5).

4.8.8 State command

Syntax: `state <UnitID> <value> [<UnitID> <value>]*`

Example: `state 3 10 4 20 5 30`

The *state* command is used to set the *state* of one or more units. It expects one or more unit-identifier/state-value pairs. The unit identifiers may be specified in any of the usual ways (see section 4.5).

4.8.9 Weight command

Syntax: `weight [<From> <To> <sitename> <value> | random [<mean> <deviation>]]*`

Example: `weight 3 5 excite 200 3 6 excite 100`

The *weight* command sets the value of a weight on a link. The first unit index is the originating unit. The second unit index is the receiving unit, and the sitename is the name of the site on the receiving unit to which the link is attached. The weight is scaled up by a factor of 1000. Thus a weight of 500 indicates a real weight of 1/2. In the floating point simulator you may use floating point values as well as integers. Random weights may be assigned, e.g. the command:

```
weight 0 1 excite random 500 100
```

will assign a weight picked randomly from the range 400 to 600.

Multiple weight settings may be given with one command, as with the *out*, *pot*, and *state* commands. The unit specifications may be given in any of the usual ways ('help UnitId' for details). The site name may be 'all', meaning all sites at the destination unit(s).

4.8.10 Reset command

Syntax: `reset`

The *reset* command is used to reset the network to some initial state. It resets the potential and state of all units to their original values (as specified when they were made with *MakeUnit*). It also resets the outputs of all units to be zero. It does not reset weights or any other parameters.

This command has been somewhat superseded by the *checkpoint* and *restore* commands (see sections 4.10.1 and 4.10.2).

4.9 Set commands

Sets may be manipulated from the command interface as well as from within user code (see section 2.9). Sets provide a way of imposing some structure on what is essentially an amorphous mass of units.

4.9.1 Addset command

Syntax: `addset <set name> <UnitId>`
Example: `addset still-competing W-T-A`

The *addset* command adds one or more units to a set. The units may be specified using any of the normal methods (see section 4.5). If the set does not already exist it is created.

4.9.2 Remset command

Syntax: `remset <set name> <UnitId>`
Example: `remset still-competing W-T-A[0] - W-T-A[4]`

The *remset* command removes one or more units from a set. The units may be specified using any of the normal methods (see section 4.5).

4.9.3 Deleteset command

Syntax: `deleteset <set name>`
Example: `deleteset still-competing`

The *remset* command deletes a set. All units in the set are removed from the set and the name is reset to be unused.

4.9.4 Unionset command

Syntax: `unionset <answer set name> <set A> <set B>`
Example: `unionset winners still-competing-A still-competing-B`

The *unionset* command assigns the union of the second and third sets to the first set. All units which are in either the second set or the third set or both are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the union of the second and third sets are removed from it.

4.9.5 Intersectset command

Syntax: `intersectset <answer set name> <set A> <set B>`
Example: `intersectset winners still-competing-A still-competing-B`

The *intersectset* command assigns the intersection of the second and third sets to the first set. All units which are in the second set and the third set are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the intersection of the second and third sets are removed from it.

4.9.6 Diffset command

Syntax: `diffset <answer set name> <set A> <set B>`

Example: `diffset winners still-competing-A still-competing-B`

The *diffset* command assigns the difference of the second and third sets to the first set. All units which are in the second set but not the third set are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the difference of the second and third sets are removed from it.

4.9.7 Inverseset command

Syntax: `inverseset <answer set name> <set name>`

Example: `inverseset losers still-competing-A`

The *inverseset* command assigns the inverse of the second and third sets to the first set. All units which are not in the second set are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the inverse of the second sets are removed from it.

4.10 File commands

It is possible to save the *structure* and/or *state* of a network to file, and load the saved file in later. It is also possible to make a log file of the session, both user input and simulator response, and to read in a file of simulator commands. The simulator automatically makes a log file of the commands typed in for each session, and at the end of the session (see section 4.11.1) asks if it should be saved.

4.10.1 Checkpoint command

Syntax: `checkpoint [<file name>]`

Example: `checkpoint conceptlearn`

The *checkpoint* command is used to save a the state of a network to file. The state consists of the values of unit parameters (e.g. weights, potentials, etc), the current sets and unit state names. The network state may be restored with the *restore* command (see section 4.10.2). Related commands are *save* and *load* (see sections 4.10.3 and 4.10.4), which save and reload the structure (pattern of units and links) of the network as well.

A name does not have to be specified for the checkpoint file, if it is not the simulator will construct one involving the UNIX process id number. The convention is that all checkpoint file names are of the form *name.chk.n*, where *n* is an automatically incremented integer.

For example:

```
-> checkpoint conceptlearn
chk file name [default: conceptlearn.chk.1] >
saving state
.....
->
```

Here the simulator appended *.chk.1* to the specified file name, and asked for confirmation. The row of dots indicate the checkpoint process, each dot representing ten units having been checkpointed.

4.10.2 Restore command

Syntax: `restore <file name>`

Example: `restore conceptlearn.chk.1`

The *restore* command restores the state of a network from a file made with the *checkpoint* command. This command simply restores the state of the network (i.e. the weights, potentials, etc), it does NOT rebuild the network. That means the network must already have been built. The command will check that the file used for restoration was made from the same simulator with the same network. If the simulator has been recompiled, but has had the same network constructed in it, the warning message may be safely ignored. The simulator will also issue a warning if the checkpoint file was made during a previous session.

For example:

```
-> restore conceptlearn.chk.1
restoring unit state.....
restoring set names
restoring state names
state restored.
->
```

4.10.3 Save command

Syntax: `save [<file name>]`

Example: `save conceptlearn`

The *save* command is used to save a the *structure* and *state* of a network to file. The structure of the network is the units, sites and links and associate names and functions. The state consists of the values of unit parameters (e.g. weights, potentials, etc), the current sets and unit state names. The saved network may be reloaded into an empty simulator with the *load* command (see section 4.10.4). Related commands are *checkpoint* and *restore* (see sections 4.10.1 and 4.10.2), which save and reload the state of the network only.

A name does not have to be specified for the save file, if it is not the simulator will construct one involving the UNIX process id number. The convention is that all save file names are of the form *name.net.n*, where *n* is an automatically incremented integer.

For example:

```
-> save conceptlearn
net file name [default: conceptlearn.net.1] >
saving name table ...
saving units.....
saving state
.....
->
```

Here the simulator appended *.net.1* to the specified file name, and asked for confirmation. The row of dots indicate the save process, each dot representing ten units having been saved.

4.10.4 Load command

Syntax: `load <file name>`

Example: `load conceptlearn.net.1`

The *load* command loads the *structure* and *state* of a network from a file made with the *save* command. This command constructs the network (i.e. makes units, sites and links) and restores the state of the network (i.e. the weights, potentials, etc). The simulator should be empty: no network building functions can have been called. The command will check that the file used for loading was made from the same simulator executable, and issue a warning message if the executables are different. If the simulator has been recompiled, with the same functions available, the warning message may be safely ignored.

For example:

```
-> load conceptlearn.net.1
loading units..... units loaded
restoring unit state.....
restoring set names
restoring state names
state restored.
Done!
->
```

4.10.5 Log command

Syntax: log [<on/off>]

Example: log on

The *log* command is used to switch logging on and off. If logging is on, then everything typed at the keyboard, and everything sent to the screen by the simulator is saved in a log file. When logging is switched off, the current log file is closed. If logging is switched on again, the previous log file may be appended to, or a new log file may be started. For example:

```
-> log on
log file name [default: run2462 log.1] >
-> log off
-> log on
log file name [default: run2462 log.1] >
Overwrite file run2462 log 1 (y,n,a[ppend]) ? a
->
```

The convention is that all log files are of the form *name.log n* where *n* is an automatically incremented integer. The simulator constructs a logfile name from the UNIX process id number (as above), and asks for confirmation or another file name. If a network is constructed with debugging and Auto-Fix switched on, and a log file is kept, then errors may be accumulated in the file for one-time correction, just as compilers for conventional languages accumulate errors before aborting the compilation.

4.10.6 Read command

Syntax: read <file name>

Example: read conceptlearn.cmd.1

The *read* command causes the simulator to read commands from a file. The commands in the file should have exactly the same format that commands typed on the keyboard have. *read* commands may be nested - in other words a file of commands that is being read may contain a read command to commence reading a different file, and return to reading the current one when the end of the new one is reached.

A command file containing the commands typed at each session is created, and at the end of the session the simulator asks if it should be deleted or kept (see section 4.11.1). This enables easy rerun of a session. For instance, the following command script is used to control the simulation for the example in section 6.

```
call build map
show on
show pot 1
show set + change
pipe off
pause on
async
printpause are you ready
go 10
```

4.11 Miscellaneous commands

4.11.1 Quit command

Syntax: quit

The *quit* command is used to exit the simulator (or to exit a higher level interface - see section 4.3.2). When exiting to the UNIX shell, the simulator will ask if the file of commands typed during the session should be kept. If the answer is yes, it will ask for a file name. The convention is that command file names are of the form *name.cmd.n*. For example:

```
-> quit
Save command file (y,n) ? y
cmd file name [default: run2462.cmd] > conceptlearn.cmd.1
%
```

From higher level interfaces, the *quit* command will return to the next lowest interface (if all errors have been fixed - see section 4.3). Typing control-D to the prompt at any level interface will result in immediate exit to the UNIX shell (after possibly saving the command file).

4.11.2 Print command

Syntax: print <message>

Example: print finished reading in concepts

The *print* command simply prints the message it is given. For example:

```
-> print finished reading the message
finished reading the message
->
```

This command is intended for use in command files. When a command file is being *read*, the commands are not echoed on the screen. This command can be used to indicate significant stages in the simulation specified in the command file.

4.11.3 Printpause command

Syntax: printpause <message>

Example: printpause finished reading in concepts

The *printpause* command is a combination of the *print* and *pause* commands. It prints the message and then the simulator pauses for the user to indicate to continue. For example:

```
-> printpause finished reading in concepts
finished reading in concepts
PAUSE - any char to continue <user hits a key here>
->
```

Like the *print* command, this command is intended for use in command files. The command file can display interesting units, and then *printpause* while the user examines the data, before continuing. During the pause, the user can even type control-C, thus entering the interrupt interface (see section 4.3.5) whence the network may be examined in detail, and when the interrupt interface is exited, the command file will continue to be read.

4.12 Abbreviated commands

Several commands have abbreviations:

Abbreviation Command

d	disp
e	echo
g	go
l	list
o	out
p	pot
q	quit
s	state
sh	show
w	weight

In addition some of the terms used in commands have abbreviations:

a	all
c	connections (equivalent to link)
def	default
u	unit

4.13 Constructing a network from the command interface

Using the *MakeUnit*, *AddSite*, and *MakeLink* commands it is possible to construct small simple networks from the command interface. The first *MakeUnit* command will cause space for 200 units to be allocated, or the *AllocateUnits* command may be used to explicitly create the space. For example, the following script file makes a network of four units, linked in a ring, and names them as a vector.

```
MakeUnit mytype UFsum
MakeUnit mytype UFsum
MakeUnit mytype UFsum
MakeUnit mytype UFsum
AddSite 0 mysite SFweightedsum
AddSite 1 mysite SFweightedsum
AddSite 2 mysite SFweightedsum
AddSite 3 mysite SFweightedsum
MakeLink 0 1 mysite 1000 0 NULL
MakeLink 1 2 mysite 1000 0 NULL
MakeLink 2 3 mysite 1000 0 NULL
MakeLink 3 0 mysite 1000 0 NULL
NameUnit NOVICE vector 0 4
```


5 Advanced Programming Features

We have tried to ensure that the functions and facilities described in the preceding sections cannot corrupt the network structure, or cause the simulator to dump core, even if misused. However, such security imposes limitations that experienced users will find irksome. Consequently the simulator has purposely been designed so that an advanced programmer is limited as little as possible. This means that all internal data structures apart from the Name Table are available to user code. The following sections give an overview of unsecured features, which are described in detail in the Advanced Programming Manual. Details of features that are commonly used in user code appear here.

5.1 Simulator Variables

Many variables used by the simulator are accessible to user code. Modifying these should be done with care. The variables which are often used in user programs follow. Most are documented in the Advanced Programming Manual.

Unit * UnitList	pointer to the unit array
int NoUnits;	number of units made so far
int LastUnit;	index of last unit that there is space for
char ** StateNames;	array of state names, indexed by state value
int NoStates;	maximum number of states with names
int StateCount;	actual number of states with names
char ** SetNames;	array of set names, indexed by set number
int LastSet;	maximum number of sets allowed (currently 32)
int NoSets;	actual number of sets used
int NoLinks;	number of links made so far
int Clock;	the system clock
int Pause;	if TRUE, Pause after every Show
int EchoStep;	print message after this number of steps
int Logging;	TRUE if a log file is being created
FILE * LogFile;	file pointer for Log file
int Show;	TRUE if showing is switched on
int ShowStep;	display units at this number of steps
int ShowPot;	display units with potential over this limit
unsigned int ShowSets;	bit vector for sets to be shown

The use of these variables is documented in the Advanced Programming Manual.

5.2 Network Access Functions

There are a large number of access functions and macros that allow user code to retrieve and set values in the network data structure. The following subsections describe the kinds of facilities available, and detail functions and macros which are commonly used in user code. A much more complete description is given in the Advanced Programming Manual.

5.2.1 Display functions

The simulator functions used to display, list and show units and links can be called from user code. The piping mechanism through which these functions usually print is also available.

5.2.2 Naming

The Name Table is available for user code access, to add, delete, and modify names. In addition to the basic functions which are used to access the Name Table, the following functions are common in user code, particularly activation functions.

```
char * IndToName(u)
    int u;
```

Returns the name of the unit with index *u*, or ****NO NAME**** if the unit has not been given a name. If the name is that of a VECTOR or ARRAY, the name has the form *name[offset]* or *name[row][column]*.

```
int NameToInd(name, column, row)
    char * name,
    int column, row;
```

Returns the index of the unit with the given name. If the name is that of a VECTOR, then *column* gives offset of the unit within the vector. If the name is that of an ARRAY, then *column* and *row* give the column and row of the unit within the array. If the name is not that of a unit, or either of the indices are out of range, then the function returns -1.

More naming functions appear in the Advanced Programming Manual.

5.2.3 Simulating

Several simulation functions which closely correspond to simulation interface commands can be called from user code.

```
Reset()
```

Resets the network: sets the system Clock to zero; sets the potential and state of each unit to *init_potential* and *init_state* respectively; sets the output of each unit to zero

```
Step(count)
    int count;
```

Simulates *count* steps. Echoes and shows will be done if appropriate.

5.2.4 Modifying and Accessing the Network

There are a large number of functions which can be used to modify values in the network data structure. The ones often used are:

```
SetOutput(index, value)
SetPotential(index, value)
SetState(index, value)
SetData(index, value)
    int index, value;
```

Unit *index* is given output, potential, state, or data *value*.

```
int GetOutput(index)
int GetPotential(index)
int GetState(index)
int GetData(index)
    int index;
```

Output, potential, state or data of unit *index* is returned.

5.2.5 Unit macros

These macros access the deal with unit indices and pointers.

```
LegalUnit(index)
    int count;
```

computes TRUE if *index* is the index of an existing unit. FALSE otherwise.

```
UnitIndex(up)
    Unit * up;
```

computes the index of the unit pointed to by *up*. If *up* does not point to a unit, computes garbage.

5.2.6 Miscellaneous library functions

The following functions are in the standard simulator library.

```
SiteValue(name, sp)
    char * name;
    Site * sp;
```

If *sp* is a pointer to a linked list of site structures, such as in the *sites* field of the *Unit* structure, and *name* is the name of one of the sites in the linked list, then the function returns the *value* of that site. If no such site is found, 0 is returned and an error message printed.

5.3 Saving and reloading user data structures

The user may wish to create data structures separate from the main network data structure. Hooks are provided in the simulator to enable user-written functions to save and reload these structures when the *save*, *checkpoint*, *load* and *restore* commands are issued.

5.4 Customizing unit, site and link data structures

Each unit, site and link structure contains a field, *data*, which is for general purpose use. This field is the size of an integer or float, depending on which simulator is being used, but in any case is assumed to be the same size as a pointer. Therefore it is possible to use this field as a pointer to an arbitrary user-defined data structure. A mechanism is provided for the field to be re-defined for user code, and hooks are provided in the simulator code to call user functions to deal with displaying, saving, and loading these user-created structures.

5.5 Customizing the simulator command interface

Any user written function which has a name commencing "Cmd." will be treated as a regular command by the simulator, and will be available at all interfaces. The simulator passes an *argc-argv* structure to command functions. In addition, any command commencing "Debug.Cmd." will be available at the debug and interrupt interfaces (see sections 4.3.2 and 4.3.5), and will take precedence over level 0 commands of the same name.

5.6 Modelling Time

One aspect of neural modelling that the simulator was not designed to deal with is modelling time, for instance modelling propagation delays along fibres. By customizing the network data structure as outlined above, it is possible to adapt the simulator to model such parameters, albeit in a somewhat limited fashion. An example is given in the Advanced Programming Manual.

6 An Extended Example of network construction

In this section we provide an examples of a relatively complex network in order to introduce some of the ways in which networks may be constructed. This network is one of the examples found in the example subdirectory (usually `connect/example`, but this is site-dependent). Read the *README* file before running it.

6.1 Designing a network

The process of creating a connection network can be broken down into three related stages. First a design for the network in terms of units, links and activation functions (unit, site and link) is specified. Second a program is designed to build this network. Third the program is coded in C. We present these three stages for the Four-Color problem. If one is building a network to model some process specified at a higher level of description (e.g. a model of a cognitive process), the higher level specification must be made before a network to implement it can be designed.

6.2 The Problem: four coloring a map

The problem is to color a map using four colors so that no neighboring regions have the same color. The colors are RED, BLUE, GREEN and WHITE. We shall represent each region with four units, one for each color. The dynamics of the network are simple. Each region wants to turn on exactly one color node. If some region is shut out by neighbors' colors it may turn on some color anyway, which may force off some other region's color node. The general idea is for each region's units to inhibit each other (corresponding to the notion that a region can have at most one color), and for neighboring regions to inhibit each other from having the same color. We shall use asynchronous simulation to ensure that the search space is explored until a stable state corresponding to a correct coloring is found.

We have four units for each region. What exactly will the links be? Our units will have one site - "inhibit". Each region's units must inhibit each other - we shall accomplish this by making an inhibitory link from each unit to each of the other units in that region. Thus each region will require twelve links internally - three for each of the four units. Since it is a very strong constraint that a region have only one color, the weight on these links will be highly negative (high inhibition).

How shall we accomplish inter-region inhibition? No two neighboring regions can have the same color, so if two regions have a border, we make an inhibitory link from the color units in one region to the corresponding units in the other. For instance, if region X borders region Y, we add inhibitory links from region X's blue unit to region Y's blue unit, and vice versa. The same applies for the red, green and white units. Since we wish the network to search the space of possible colorings we must allow neighboring regions to have the same color for a short period of time, so the weight on these links will be moderate and negative (moderate inhibition)

How do we control the search through the state space? First we will say that if a unit is receiving no inhibition, it will turn on. If there is strong inhibition (e.e. from another color in the same region), the unit will remain off. If there is weak inhibition (i.e. from bordering region(s) of the same color, the unit will turn on or remain on with a probability which decreases with the amount of inhibition.

Finally, exactly what activation functions shall we use to implement the descriptions above? Since the weights are not changed, link functions can be the NULL function. The site function will simply take the weighted sum of the inputs. What weights shall we use? For the intra-region inhibition the maximum negative weight of -1000. For the inter-region same-color inhibition we shall use -100. The unit function will simply look at the inhibition arriving. If there is none, i.e. the region is not yet colored and no neighboring region is colored with the same color, then the unit will turn

on with activation and output 1000. If there is inhibition from neighboring countries (in the 100's) but not from units in the same region (in the 1000's) then with some small probability dependent on the strength of the inhibition, the unit will turn on with activation and output 1000. If there is inhibition from another unit in the same region (i.e. it is already colored) then the unit will turn off with activation and output 0.

6.3 Designing the build program

There are several things to note about the network design. The map of the regions is not specified explicitly. All links are bi-directional, i.e. if unit X inhibits unit Y then unit Y inhibits unit X with both links having the same weight. The links can be divided into two kinds, intraregion (twelve for each region), and inter-region (four for each border). These suggest some design decisions.

First we shall specify the map using a data file, which will be read in by the build program. The map file format will be:

```
<number of regions>
<region number><region number>
<region number><region number>
.
.
.
<region number><region number>
```

The first item specifies the number of regions, the remaining pairs specify the pairs of regions which border each other (each region is assumed to have been assigned a number). Thus the data file:

```
3
1 2
3 2
```

would specify a three region map with one region sandwiched between the other two, as in El Salvador, Honduras, Nicaragua. The map will be read in by the top level function, *build*.

Since the links are effectively bi-directional, we shall have a function *mur* (mutual exclusion) which creates two links, one in each direction. Since each region has the same internal structure, we shall have a function *region* to create a region. Since each border consists of mutual exclusion links between corresponding color units for the two regions, we shall have function *border* to create these links.

How shall we assign the units? Our method will be to assign them in blocks of four, one block for each region. Within each block, the first unit will be for RED, the second BLUE, the third GREEN, and the fourth WHITE. For example unit number 6 (the seventh unit as unit numbers start at 0) will be the GREEN unit for the second region.

Thus the overall mechanism for building the network will be to read in the number of regions, allocate sufficient units accordingly, make the specified number of regions (with *region*), and then read in each border and make it (with *border*).

The site for each unit will have flag NO_LINK_FUNC_FLAG set to indicate that no link functions are operative. The site function will simply be the library function SFweightedsum. The unit function will operate as described above, with the addition that it will add the unit to a set, *change* if the potential changes, and remove it if it doesn't. While simulating we may then use the *show* command to cause just the units which have changed since the last step to be displayed. In addition we will have two named states, *Change* and *Static*, which will correspond to membership or not of the set change respectively.

6.4 Implementing in C

Now that we have the design, the build program can be coded.

6.4.1 Top level *build* function

At the outermost level, the program looks like this:

```
#include "sim.h"                /* simulator definitions */
#define STATIC 0                 /* constant state names */
#define CHANGE 1

int UFcolor();                  /* unit function declarations */

/* build is the topmost network building function. It is called from
   the simulator to build the network */

build(argc,argv)
    int argc,                    /* number of arguments */
    char *argv[];               /* array of argument strings */
{
    int count,i,reg1,reg2,readstatus;
    FILE *infile;

    if(argc != 2)
    {
        /* expect build + 1 argument */
        printf("Usage: build <desc file name>\n");
        return;
    }

    infile = fopen(argv[1],"r"); /* open data file */
    if(infile == NULL)
    {
        /* if cannot open file */
        printf("build: could not open %s\n",argv[1]);
        return;
    }

    fscanf(infile,"%d",&count); /* read number of regions from file */
    AllocateUnits(count*4);      /* make units for regions X 4 colors */

    DeclareSet("change");        /* declare a set name */
    DeclareState("Static",STATIC); /* state name declaration */
    DeclareState("Change",CHANGE); /* state name declaration */

    for(i = 0;i < count;i++)      /* make the regions */
        region();

    for(i=0;;i++)                 /* make the borders */
    {
        /* exit loop with break statement */
        readstatus = fscanf(infile,"%d %d",&reg1,&reg2); /* read two regions */
        if(readstatus != 2) break; /* assume end of data - leave loop */
        border(reg1,reg2);        /* make the border */
    }

    printf("%d regions with %d borders",count,i);
}
```

The first two lines allow us to use the unix input/output facilities, and the simulator functions respectively. These lines will be present in every build program. The next two lines define constants `STATIC` and `CHANGE` to have values 0 and 1 respectively. The fifth line declares the unit function which is defined later. Finally the definition of the function *build* is given. This function is called from the simulator interface with the name of a data file as the only argument. The first if statement checks that indeed only one argument to build has been given. The second if statement checks that the argument is indeed a file that the program can read. If either of these checks fail, the function returns to the simulator with an error message.

Assuming all is ok so far, the number of regions (the first item in the data file) is read into variable *count*. Each region requires four units so *count*4* units are allocated with a call to *Allocate Units*. The set we wish to use for display, *change* is declared, and the state names "Static" and "Change" are bound to the values "STATIC" and "CHANGE". If this seems confusing, remember that the units keep their states as a number, e.g. `STATIC == 0`. By binding a name to a state value, using *DeclareState*, the simulator will display the name rather than the number.

Finally we are ready to do the real work of building the network. The first for statement makes the appropriate number of regions by successive calls to the function *region*. The second for statement repeatedly reads a border specification from the map file and makes it via a call to the function *border*, until the end of the map file is found. Once all this is accomplished, the network is in place and the user is informed how many regions and borders were made.

6.4.2 Making a region

The code to make a region is given below.

```
/* Make a region. Four units are made - one for each color. The entire
   region is named as an array, with the color macros below specifying
   the appropriate indices. They inhibit each other strongly. */

#define RED 0
#define BLUE 1
#define GREEN 2
#define WHITE 3
static char *colornames[] = {"red","blue","green","white"};

static int region()
{
    static int regionnum = 0; /* which region - static means value remains
                               between calls; it is initialized at startup
                               to 0 */

    int i,j,first;
    char buf[15];

    /* make 4 consecutive units, each for a different color */
    first = makecolor(RED); /* save index for name declaration*/
    makecolor(BLUE);
    makecolor(GREEN);
    makecolor(WHITE);

    /* name these as a single vector */
    sprintf(buf,"region%d",regionnum); /* buf contains name */
    NameUnit(buf,VECTOR,first,4); /* vector of length 4 */

    for(i = 0; i < 4; i++) /* make them inhibit each other */
        for(j = i+1; j < 4; j++)
            mux(first+i,first+j,-1000);
    return regionnum++; /* returns region number */
}
```

The first few lines are to define values for our colors. These will be used to index into the string array *colornames*, so they take values 0, 1, 2 and 3 in the same order as in *colornames*. Now we come to the code for the function *region*. It is declared static to ensure that when loading multiple object files this function is not visible to any other object file (consult a C manual if this is not clear). The first line in the function declares an integer *regionnum*, which is also static. The effect of this version of static is to make *regionnum* a permanent variable, so that it retains its value between calls to the function. At each call to the function it is incremented, so it represents the number of the region that is currently being made. The first region will have number zero.

The task of making the units for the region is done by the four calls to *makecolor*. Each call makes one unit corresponding to one color for the region, and because the calls are consecutive, the four units are next to each other in the unit array. We name them using the *NameUnit* simulator function. Finally the mutually exclusive links between the region's units are made by calls to *mux*. The weight on the links is set to -1000, as specified in the design. We return the number of the region just made, and increment *regionnum* before exiting.

Next we describe the code for the functions *makecolor* and *muz*

```
/* make a single unit representing a region color; return index of that unit */
static int makecolor(type)
    int type;          /* a string containing color name */
{
    int index;

    index = MakeUnit(colornames[type],UFcolor,0,0,0,0,STATIC,STATIC);
    AddSite(index,"inhibit",SFweightedsum);
    SetFlag(index,NO_LINK_FUNC_FLAG); /* weights don't change */
    return index;
}

static mux(unit1,unit2,weight)      /* mutually exclusive links */
    int unit1,unit2,weight;
{
    MakeLink(unit1,unit2,"inhibit",weight,0,NULL);
    MakeLink(unit2,unit1,"inhibit",weight,0,NULL);
}
```

The function *makecolor* makes a unit (*MakeUnit*), adds a site called "inhibit" to it (*AddSite*), sets the no link function flag, and returns the index of the unit made. Notice that the type of the unit is taken from the string vector *colornames* according to the color being made, and that the unit function is called *UFcolor* - this will be described below. The initial and reset state for the unit is *STATIC*. Also notice that the site function is *SFweightedsum*, a library function.

The function *muz* simply makes two links, one in either direction, between the two units, each link going to site "inhibit" and having the same weight.

6.4.3 Making borders

Making a border is a simple matter of creating mutually inhibitory links between the region units representing the same color. The function *mapunit* is a utility to get the unit index from the region number and unit color.

The border is actually made by function *border*. This calls *mux* to create the four pairs of links (one pair for each color) using *mapunit* to get the unit indices. The code appears below.

```
/* mapunit takes a region number and color and returns that unit's index */
static mapunit(region,color)
    int region,color;
{
    char buf[15];
    sprintf(buf,"region%d",region);          /*get name of region in buf */
    return NameToInd(buf,color);             /*look up index */
}

/* make a border between region1 and region2 */
static border(region1,region2)
    int region1,region2;
{
    int i;

    mux(mapunit(region1,BLUE),mapunit(region2,BLUE),-100);
    mux(mapunit(region1,RED),mapunit(region2,RED),-100);
    mux(mapunit(region1,GREEN),mapunit(region2,GREEN),-100);
    mux(mapunit(region1,WHITE),mapunit(region2,WHITE),-100);
}
```

There are faster ways of finding the index of a unit than the name lookup used by *mapunit*. For example, the expression *4*region-color* gives the correct index. However, if we decide to add new units to the network, the expression might be invalidated, but the name lookup will still work. This is an important general principle: if you wish your network to be modifiable, use the simulator functions to access data items, rather than finding them with your own code. In this case, with small maps, using the simulator functions produces no noticeable delay.

No link functions are required because weights don't change. The *NO_LINK_FUNC_FLAG* can also be set in the units flags. If it wasn't, a default, empty link function would be called for every link, for each step of simulation. This does no harm, but does take some time.

6.4.4 The unit function

The unit function *UFcolor* performs the function described in the design. Color units are either on (potential and output equal to 1000) or off (0). If there is no inhibition or *dice* returns true a unit will turn on. The macro *dice* returns true with a probability dependent on the amount of inhibition the unit is receiving, and is used to determine whether or not to switch the unit on. If the inhibition is over 1000 (another color unit in the same region is on) then *dice* returns false. If the inhibition is under 1000 (corresponding color unit(s) in neighboring region(s) are on) then the unit will turn on or remain on with probability approximately $(10 - \#conflicts)/20$.

```
/*
  dice(inhibit) has value 1 with probability: (1000+inhibit)/1999;
  Note that if inhibit is <= -1000, this probability is zero.
*/
#define dice(inhibit) ((random()%1999) < (1000 + inhibit))

UFcolor(up)
  Unit *up;
{
  int inhibit;
  int oldpot;

  oldpot = up->potential;          /* remember old potential */
  inhibit = SiteValue("inhibit",up->sites);
  if(inhibit >= 0 || dice(inhibit))
  {
    /* unit on */
    up->potential = 1000;
    up->output = 1000;
  }
  else
  {
    /* unit off */
    up->potential = 0;
    up->output = 0;
  }

  /* change state and set membership if necessary */
  if(oldpot != up->potential)
  {
    AddSet("change",UnitIndex(up));
    up->state = CHANGE;
  }
  else
  {
    RemSet("change",UnitIndex(up));
    up->state = STATIC;
  }
}
```

The function *UFcolor* defines the behavior of the color nodes, and thus, of the whole network.

Units are in one of two states: *STATIC* and *CHANGE*. The state does not affect their behavior, but it does make it easier to see what is happening when watching unit lists scroll by. Several steps in a row with no units in a change state probably means the network has settled on a solution.

UFcolor also dynamically updates membership in the set "change" so that it contains units which have changed state this step. This set is used as a show set, so units which have just changed will always participate in a show.

6.5 A command script to demonstrate the network

An important way of controlling the simulation is the use of command scripts. The following script has all the commands necessary to start and run the four color network.

```
call build map
show on
show pot 1
show set + change
pipe off
p-use on
async
printpause are you ready
go 10
```

The first command calls the function *build* with map file *map*. This causes the network to be constructed. The three show commands together mean that we will see only those units which are on or have just gone off. Turning the pipe off means that output to the terminal will not go through the more filter. Turning pause on causes the simulator to wait for a prompt after each step. The simulation must be run asynchronously or the whole network will oscillate: all on, all off, all on, ... The *async* command puts the simulator in asynchronous mode with all units simulated at each step. The *printpause* command prints the message and waits for a prompt. The final command tells the simulator to run for 10 steps.

Command scripts can considerably ease the burden of simulating. It is easy to create script files with slight variations in the simulation parameters and control. Feeding in a lot of information to a network can be done as well, for instance for initialization.

7 Floating Point version

The floating point version of the simulator uses floating point values for *potentials*, *outputs*, *site values*, *weights*, *link values*, and *unit, site and link data* fields. The floating point version is created by specifying the *-f* flag to *makesim* (see section 3). If this flag is given, user code will be compiled with the flag *-DFSIM* used (see the man page for the C compiler, */em cc /em*, for details of the *-D* flag). Thus user code may use conditional compilation to automatically be recompiled for integer/floating point simulation. The type *FLINT* is defined in the user compilation environment to be a float or an *integer* depending on whether the compilation is for the floating point or integer simulator. More details are given in the Advanced Programming Manual.

8 Acknowledgements

The first version of the simulator was designed and implemented by Stephen Small, Lokendra Shastri, Gary Cottrell and others. Mark Fenty converted it from LISP to C and made extensive changes. Mark has been a constant source of Good Advice during development and upgrade to release quality software. As always, Jerome Feldman provided inspirational comments when they were most needed.

The Rochester Connectionist Simulator
Volume 2:
Graphics Interface User Manual

Kenton Lynne
Dept. of Computer Science
University of Rochester
Rochester, NY 14627

April 15 1987

Contents

1 Background	2
2 Getting Access to GI Functions	2
3 Using the GI Tool	3
4 The Command Panel	4
5 The Message Panel	4
6 The Mode Panel	5
7 The Control Panel	6
7.1 Lower control panel - running the simulation	6
7.2 Main Mode - laying out the network	6
7.3 Link Mode - checking the connections	9
7.4 Text Mode - displaying text	10
7.5 Draw Mode - line drawings	11
7.6 Custom Mode - customizing mouse buttons	11
8 The Display Panel	13
9 The Info Panel	15
10 GI command interface	16
10.1 Placing units on the graphics display	16
10.2 Drawing lines or boxes and adding text	18
10.3 Moving and deleting objects	18
10.4 Simulating and updating the graphics display	19
10.5 Redisplaying	19
10.6 Displaying unit details	19
10.7 Mapping mouse buttons	19
11 Advanced Features	21
11.1 Creating and using your own icons	21
11.2 The programming interface for GI functions	22
11.3 Using the log file	23
12 Multiple unit views	25
13 Performance Hints	26
14 Future Directions	27

1 Background

The Graphics Interface (from now on called GI) was developed as an extension to the Rochester Connectionist Simulator for use on Sun graphic workstations. It provides a graphic output display for networks created by the Rochester Connectionist Simulator making it possible to observe the behavior of the network as it runs. Once your network has been built by the simulator, GI gives you a display panel upon which can be arranged graphic symbols (icons) representing particular aspects (potential, output, state, links, etc) of units of your network that you want to observe. As simulation steps are run, the appearance of these icons will change as the values of their selected aspect change. Thus you can visually observe the dynamic overall behavior of your network (or part of it) as it runs, which can be very useful in getting an intuitive feeling for what the network is doing. You can also draw and write text and line drawings on the display panel for documentation of your network. Remember that GI is strictly a "read-only" interface: that is, anything you do with the graphics has no affect on your network itself and the base simulator is (almost) completely unaware of GI's existence.

Note: All references to "simulator" in this manual refer specifically to, and only to, the software package known as the *The Rochester Connectionist Simulator* and the assumption is that you already have a working knowledge of it. So if you are unfamiliar with the simulator itself, you should probably read TR (forthcoming) which explains its basic concepts and operation, or the document *The Rochester Connectionist Simulator: User Manual*.

2 Getting Access to GI Functions

The GI is linked into the simulator itself at "makesim" time automatically. If you do *not* want the GI graphics package you should use the "-ng" flag with your makesim command. Because Suntool graphic tools pull a lot of library routines into their objects, the final load objects tend to be quite large, usually between 300 and 500 KB (depending on whether you compile with the -g option). You will probably want to delete these objects once you are finished with them if disk space is a critical resource as it is at the University of Rochester.

RESTRICTION: If your own simulator code makes use of external variables you must be aware that name clashes are possible between your code and the GI functions since they are linked together as one object. All GI external variables and functions have names beginning with the prefix "gi.", so you would be wise to avoid naming any of your external objects using this prefix.

3 Using the GI Tool

Once you've made your simulator with "makesim" you can run your simulation session with the GI interface. Simply type in the name of your simulator object as usual. You will notice a message when the simulator comes up "building Graphic Interface tool - please wait". It will take a few seconds and then you should see the GI tool come up on the righthand side of your screen (see Figure 1). You can use any of the Suntool "-W" flags following the simulator command if you wish to customize aspects of the tool. For example, the command:

```
mysim -Wp 0 100 -Ws 1000 800
```

would bring the GI tool up on the extreme left-hand side of the screen, 100 pixels from the top and sized at 1000 pixels wide by 800 pixels high. See the Suntools documentation for a complete list and explanation of all the -W options available. Note: if you are using the optional saved simulator file parameter, it must appear before any of the -W flags.

Once the GI tool has appeared on the display, you will notice it consists of six separate (but interrelated) panels:

- The **info** panel is the top panel of the display and is used for showing detailed textual information about specific units in your network.
- The **mode** panel is the wide short panel immediately below the info panel and is used both for changing "modes" (described below) and for turning logging on or off (also described below).
- The **display** panel is the mostly large blank panel on the left side of the window just below the mode panel. It is the canvas on which the graphical representation of your network will be displayed.
- The **control** panel is the complicated looking panel with all the buttons and prompts and is just to the right of the display panel. This panel is the primary way you will interface to the GI functions in order to control the graphical expression of your network on the display panel.
- The **message** panel is the thin panel directly below the display panel. It is used solely for displaying errors, warnings or informational messages.
- The **command** panel is just below the message panel and is the bottom most panel of the GI tool. It is your primary interface with the base simulator and essentially takes the place of its command interface.

Like most Suntools, in order to work within a particular panel, you need to move the cursor into it. Successful operation of GI requires understanding how these panels operate and how they relate to one another. A detailed description of the operation of each panel is contained in the next sections.

4 The Command Panel

The command panel is the the bottom leftmost panel of the GI tool and usually the first one you will use. It consists of a prompt "-->" which looks very much like the standard simulator prompt for good reason: it essentially is the simulator prompt. That is, any simulator command that you type followed by a return is sent unchanged to the simulator command interpreter and executed. For example, if the first thing you normally do is build your network with "call buildmynet", that is exactly what you would type into this prompt. The simulator will process your command, and once it returns, the command will move up to the next line and you can type in another command. The only difference between using this command line and the one on the standard simulator is that any text output generated by the simulator (for example on a "list" command), will show up in the original window the simulator was started in, not in the GI tool window. Thus if you plan to use simulator commands for generating displayed output, you should set up your Sun windows so that you can see both the GI tool and the original simulator window at the same time. However, if the simulator command generates an error message from the simulator, the error will appear in the GI message panel as well.

5 The Message Panel

The message panel is the simplest because the only interaction you will have with it is to read text that has been put there by GI for your information or befuddlement. Mostly (hopefully) they will be confirmational messages like "Show command successful". Sometimes there will be warning messages to inform you of something that happened that you may not be aware of (such as that a unit has been "displayed" off the viewable screen) but may be OK anyway. At other times there will be error messages informing you with utter clarity as to what went wrong and maybe even a clue on how to fix it. Hopefully, very seldom will you see obscure looking messages like "undesignated <what> in get_unit procedure" which are indications of a program failure. If you run into any of these consistently, please let us know so we can try to figure out the problem. You should also see any error messages resulting from commands sent to the simulator in this window.

6 The Mode Panel

The mode panel is right below the info panel and controls two separate functions. The left side of the panel has the word **MODE:** followed by the choices "Main", "Link", "Text", "Draw" and "Custom" with one of them reverse-imaged. Clicking over "Mode:" or one of the choices will change modes accordingly. Different modes allow you to do different operations on the display panel. You will notice that switching from one mode to another results in a different set of prompts being displayed on the control panel. The mouse actions performed in the display panel will also change based on the current mode. A hint as to what the three mouse buttons do will appear in the message panel each time a mode is selected. Briefly, the different modes are used for the following functions.

- **Main mode** is used primarily for setting up and displaying the units in your network on the display panel. In main mode the control panel will have prompts that pertain to what, how and where you want the units in your network displayed.
- **Link mode** is for interactively examining and verifying the topology of your network. That is, in Link mode the icons being displayed are always showing connection strengths between units rather than some other aspect such as their output, potential, state, etc.
- **Text mode** is for placing printable ASCII characters on the display primarily for documentation and publication purposes. The control panel prompt asks for a font which allows a variety of type faces and sizes to be used on the display.
- **Draw mode** allows you to draw boxes and other straight line objects on the display for documentation and aesthetic purposes.
- **Custom mode** enables you to set the mouse buttons to specific commands that can be executed while the mouse is on the display panel. These commands can have symbolic arguments that are filled in at execution time based on where the cursor is located in the display panel.

More detail on how to work in these modes is contained in the following section: *The Control Panel*.

The right side of the mode panel contains two fields for controlling the logging of commands. **LOG:** is a switch which can be set either "On" or "Off" by clicking over it with the mouse and indicates whether you want GI and simulator commands you issue to be logged to a file. If logging is on "On" then actions that affect the display screen will be turned into commands and written to the file named in the prompt just to the right of the switch. The default log file used is named "gi.log" but of course you can change this to any file name you want. When logging is turned off, any commands that had been written to the log file are immediately flushed into the file and can thus be read in immediately if desired. If you then specify another log file and turn logging back on, then the original log file will be closed and the new log file will be opened for write (deleting its old contents) as soon as the next command completes. However if you don't change log file names, and switch logging back on and off, then the log file will accumulate commands that are issued whenever logging is on.

The reason you might want to log commands is that you can then "replay" your simulation session back later (via the simulator "read" command) thus saving yourself the trouble of setting up the network display. You can also edit the log file to "weed out" or change the way the session will be reenacted. Of course to do this you will have to understand the syntax of the simulator commands - which we assume you are already familiar with - and the GI commands which are documented in section 10. More detail regarding the log file is contained in section 11.3.

7 The Control Panel

The control panel is located on the right side of the GI tool and is normally useful only after you have built your network (presumably by interfacing to the simulator through the command panel). Through interacting with it you set up the display with the graphic representation of your network, run the network and do other miscellaneous functions such as writing the display image to a file or placing text or line drawings on the display. The control panel is divided into an upper and lower part. The upper part (above the Clock and Origin messages) is used primarily for setting up the display panel the way you want it to look, and will have different prompts displayed depending on what mode you are in. (See previous section on the mode panel). The lower part is used mostly after the network has been laid out for actually running and watching the simulation run. It looks and operates the same in every mode and we'll describe it next.

7.1 Lower control panel – running the simulation

The buttons on the lower part of the control panel are most useful during and after a simulation run. The GO button is for actually running some number of steps of the simulation once the network has been laid out on the display screen. It has two associated prompts, one ("number steps:") specifies the number of steps to run, and the other ("update steps:") indicates how often to update the display while the simulation is running. For example setting "number steps:" to 20 and "update steps:" to 4 will cause 20 simulation steps to be run with the display being updated every 4 steps.

The DUMP button is used to save the actual display panel image in a raster file. The default name is "gi.image" but you can change that to any name you wish.

The RESHOW button simply rewrites everything onto the screen (from scratch, so to speak), so if in moving things around, the display has somehow gotten messed up, RESHOW should put things right again. It has a prompt: ":" which defaults to the current Origin. If you would like the display to show a different portion of the display space, change the value to the desired origin before pressing RESHOW and the display will be translated appropriately using the new origin (which will then become the current origin). The relationship between display space and the origin, is explained in detail in section 8.

The QUIT button does exactly that: ends without recourse the simulation session and returns to the shell. Make sure you've saved everything you need to before pressing this button since it gives no second chances.

7.2 Main Mode – laying out the network

When you are in *main* mode (see Figure 1), the upper part of the control panel will have three buttons representing the three basic commands that are used for initial layout of your network on the display panel: SHOW, CHANGE and ERASE. They all act by taking the appropriate parameters from the the prompts (WHO, HOW MANY, WHERE, etc) and then performing the requested action on the display screen.

The SHOW command is used to specify how and where to display units that are **not** already currently displayed. The ERASE command is used to erase units from the display (**not** from your network) that currently **are** displayed. Thus ERASE is the functional inverse of SHOW. The CHANGE command is used to change some representation of units that **are** already displayed. CHANGE is sort of a combination (or shortcut) of the ERASE and the SHOW commands. That is, it acts as though the specified units were first ERASEd and then SHOWn again with (possibly) different attributes.

The prompts (WHO, HOW MANY, WHAT, etc) need to be filled in appropriately before pressing any of the command buttons. Note that defaults are set up in all the prompts. In fact, once you've built your network you can immediately left click over the SHOW button which will execute the default SHOW command thereby laying out your entire network in a default fashion. However, for your specific purposes this default layout may be somewhat inappropriate. In order to specify the exact way you want your network to appear on the display panel, you need to understand the semantics of the prompts and how they apply to each command:

- WHO (along with HOW MANY) determine which units in your network will be SHOWN, CHANGED or ERASED. The WHO part specifies the beginning unit (if HOW MANY specifies more than one) in an ordering of units by unit index. There are four ways to specify a unit in the WHO prompt - By unit index (note the default is "0" indicating the first unit in any network), by unit name, by unit type or by set name. These are all things you can specify via functions in the basic simulator within the C code that you wrote to build your network. For convenience in starting over, the ERASE command will allow you to specify "all" for WHO which will erase everything that has been displayed.

- HOW MANY specifies the number of units the command is to be applied to, beginning with the unit specified in the WHO prompt. This can be either a decimal number or the word "all" which means all units matching the WHO prompt.

Note: Depending on what the command is, the beginning unit specified by WHO is handled slightly differently. For SHOW, the WHO unit will be the first unit (starting from unit 0) that matches a unit which is *not currently displayed*. So if, for example, your network has 100 units of type "input", then setting WHO to "input" and HOW MANY to "10" and pressing SHOW will display the first ten of them. Pressing the SHOW button again will display the next 10, and so on until SHOW can't find 10 units of type "input" that aren't already displayed (which will then result in an error message). The ERASE command sort of works in reverse. Once all 100 input units are displayed (and setting WHO to "input" and HOW MANY to "10") ERASE will erase the first 10 displayed, pressing ERASE again will erase the next 10, and so on until ERASE can't find 10 "input" units that are still displayed. CHANGE always works on the 1st unit it can find that is displayed. Thus setting WHO to "input" and HOW MANY to "10" and pressing CHANGE multiple times will always affect only the first 10 "input" units displayed.

- WHAT specifies two things: the "aspect" of the unit you wish displayed (Potential, Output, State, Data, Link/in, and Link/out) and the expected range of values that you expect that aspect to take on during the simulation. The desired aspect is selected by clicking left over it (which becomes reverse imaged) and you specify the range by typing into the "from:" and "to:" prompts. Note that the default is to display the unit potential over a range of -1000 to 1000. The meaning of "Potential", "Output" and "State" should be self-evident. "Data" refers to the data field contained in each unit. Link/in and Link/out specify that the aspect of the unit you want displayed is the link weight of that unit *to* (Link/in) another (target) unit, or the weight *from* (Link/out) another (target) unit. Note that when you select Link/in or Link/out as the aspect another prompt appears labeled "target:" This is where you specify the target unit in the same manner that you specified WHO. Note, however that the target unit is just one particular unit. Thus if you specify a set name or type, the target will be just the first one found. The safest way to specify the target to make sure you get just what you want is by unit index or unit name. If the target happens to already be on the display screen, clicking the right mouse button over it will automatically copy that unit's index to the "target:" prompt for you. If you specify just a unit in the target field, GI will use the first matching link it can find for that target. However if there is more than one link between the

target and a unit you may want to specify a particular link. You can therefore add a "site" designation to the target as well. The site designation is simply the site name where the link is attached and you specify it by appending the site name to the unit separated by a slash ("/"). Thus if you were interested in the link between target unit 43 at site "special" you would put "43/special" into the target prompt. (If you have multiple links between two units all at the same site you are out of luck: there is no way in GI to distinguish among them).

- HOW specifies what kind of graphic object(s) you want the unit aspect displayed as. There are six choices: the first five are polygons and the last a dark square, is a grey-scale icon. You select the one you want by clicking left over it which moves a horizontal bar beneath the selected choice. If you don't like any of the choices presented, there is a seventh choice (on the next line) designated by an icon with a "?". If you select the "?" icon, you are prompted to fill in the "name:" of your own icons. (see the section "Advanced Features" for details). The default choices given are only the prototype of what the unit will actually look like: the actual appearance of the unit during the simulation will depend on its current value for the aspect selected for it. The icons shown are what the unit will look like when and if it reaches the maximum value for its selected aspect. The way the value of the unit aspect is shown for the five polygons is by the two dimensions of size and shading. Each shape has 20 different icons associated with it and based on the particular value of the unit aspect at the time, one of those shapes is selected to represent the unit and is then displayed. What actually happens is that the range is divided into 20 evenly sized subranges and whatever subrange the value happens to fall into, that corresponding icon is displayed. (If the value falls outside the range, it is treated as though it were the maximum or minimum of the range). The 20 icons are ordered such that the maximum value in the range corresponds to a large, light colored polygon, and as the values decrease the polygon becomes smaller and smaller until at the middle of the range (normally 0) the point is reached at which the polygon is at its smallest. As the value gets smaller (normally more negative) the icon becomes dark and begins to get larger until at the low end of the range it appears as a large dark polygon. The exception to the above rule applies to the grey-scale icon. At its largest value it is a dark rectangle, and decreases by becoming progressively lighter until at its minimum value it is almost a completely white space it however, does not change size. By the way, if you prefer the reverse (have the dark polygons or lighter grey-scales represent larger values) simply switch the numbers in the "to:" and "from:" prompts.

- WHERE is used to control where on the display panel the aspect of the unit you've selected will appear. As such it is only looked at by the SHOW command. There are several things that need to be specified about the positioning of the icons. They all require that you understand a little about the geometry of the display panel and how it is referenced.

The objects on the display panel are positioned by x and y coordinates with 0.0 being the upper lefthand corner of the panel with x becoming increasingly positive as you move to the right and y becoming more positive as you move down. This is in accordance with Sun conventions for window geometry. On the display panel you may notice an odd looking "X"-like object which from now on will be referred to as the "marker". If you look at the prompts "start x:" and "start y:" you will notice that they have coordinates already in them. These happen to be the coordinates of the marker on the display panel. Notice that if you change these coordinates, the marker will correspondingly change position. Similarly when you move the marker around (by clicking left on the display panel where you want it to move to) the coordinates will automatically change to reflect its new position. This synchronicity between these prompts and the marker comes in handy when laying out units of your network since the "start x" and "start y" prompts indicate where the first unit will be positioned on a SHOW command. If only one unit is going to be displayed, then the rest of the WHERE prompts are irrelevant. However if a number of units are going to be displayed, the other WHERE

prompts are used to specify how the whole group are to be laid out. The underlying strategy is to lay out multiple units in a rectangular fashion in rows and columns, in "reading order", that is, left-to-right and top-to-bottom. The "start x:" and "start y:" designate the top left of the rectangle, "space x:" and "space y:" specify how many pixels to leave between each column and row respectively, and "units per row:" tells how many icons to put in each row. If "units per row:" is set to max, SHOW will put as many all the icons in one row even if that means some of the icons will be off the display panel. Be assured however that even though the units are not presently shown, they are they are still there and in the next section you'll learn different ways to make them visible. If you want the units displayed in a right to left or top to bottom order, simply use negative values for the "space x:" and "space y:" prompts, respectively. Note that you can make the units "overlap" by specifying small (less than the icon size) negative spacings. However, GI does not guarantee that the results will be pretty.

Once appropriately filled in, clicking left over the SHOW, CHANGE or ERASE buttons will perform the specified command. It useful to fool around with these commands in order to get a feel for how to arrange and rearrange the units. An important thing to remember is that the SHOW command uses all the prompts, the CHANGE command uses all the prompts except WHERE, and the ERASE command only uses WHO and HOW MANY. Remember that no amount of messing around with the display will have any effect on the network itself: in a sense GI is "read only" as far as the actual network is concerned.

7.3 Link Mode - checking the connections

Selecting Link mode on the mode panel will put you in link mode (see Figure 2) Link mode was designed for the single-minded purpose of making it easy for you to tell how your network is connected. No matter what aspect your unit icons were displaying in the other modes, in link mode all icons are always tracking the weight of links from or to some one other unit called the *target* unit. Link mode is as if you CHANGED all the units displayed to either Link/in or Link/out to or from a particular target unit. The advantage to link mode is that the original definitions of your units are not lost. That is no matter what you do in link mode, switching back to main or any other mode, will restore the units to what they were before you entered link mode. Thus think of link mode as a temporary escape from the "usual" definitions of the unit icons to one where only links are displayed. Here's how to use it.

Notice that the control panel has three prompts in it: **TARGET**, **HOW** and **DIRECTION**. The idea is to pick a unit as the "target" in much the same way as you do in main mode, either by clicking the right mouse button over the target unit or by typing the target unit name (or index, type or setname) into the TARGET prompt. You can add a site name to the TARGET prompt by appending it to the unit separated by a "/". Once you pick a target (by either clicking or pressing return in the TARGET prompt) all the other icons on the display will change to show the weight of their link (if any) to that unit. The other two prompts are for controlling how the links are shown:

- **DIRECTION**: which way the link is supposed to go. Link/in indicates that each unit on the display will show the weight of the link *from* the unit *to* the target; Link/out specifies that each unit will show the weight of a link *to* the unit *from* the target.
- **HOW**: what kind of icon and using what range of values. For choice of icon, the familiar polygonal ones are available as well as the default one labeled "same". If one of the polygonal ones are chosen, then all the displayed units will change to that one icon shape (only during link mode, of course). Selecting "same" means to use the same icon shape for each unit that was used in main mode. Range just indicates what the expected link values will be so as to proportion the icon changes appropriately among subranges.

7.4 Text Mode – displaying text

There may be times that for documentation or publication purposes you may want to put text on the display panel along with the network. Selecting "Text" on the mode panel puts you in text mode which allows you to do this (see Figure 3). Once in text mode you simply click left anywhere on the display panel where you want your text to start. A black rectangle will appear meaning you can start typing characters and they will then appear on the display. You can use the backspace key to fix errors and the Return key will put you on the next "line". However each line of text you enter will actually be treated as a separate text "object" that can be manipulated (ie. moved or deleted) independently. Also any significant mouse action, such as leaving the window or pressing another button, will create a separate object for the text entered so far and you will have to click left again to set the position for another text string. The longest possible single text object is 80 characters. If you type in a string longer than 80 characters, GI will automatically break it up into two or more text strings. You will normally be unaware that this happened unless you try to move or delete the string in which case the fact that it is not really a single object will become apparent.

The control panel in text mode has just one prompt – specifying the text font. Thus you can put text on the display in a variety of fonts (in fact each text object can be in a different font). You have to remember to specify the font **before** you set the text position as leaving the window to change font will necessitate setting the text position again. Also once you start typing in a text object, there is no interactive way (yet) to change its font. The default font is just your workstation default font and will be used if you don't specify anything or the font you specify cannot be found. Just about any font the system supports can be used. The default directory used is `~/usr/lib/fonts/fixedwidthfonts/`, so if you want a font in that directory you only need specify the filename of the font. For fonts in other directories you have to fully qualify the font file name. By the way, the **FONT** prompt has three lines: should you run out of space on the current line, typed in characters will automatically be continued on the next. A warning regarding using variable pitched fonts: The Sun 2.0 font structure did not allow efficient manipulation of variable pitched fonts (this was changed in 3.0). Thus selecting a variable pitched font will produce funny-looking spacing while you are typing the text in. However a **RESHOW** will put it back together with the proper spacing.

Text objects can be moved about the display manually by clicking the middle mouse button down over the text object. This will cause the cursor to change into a "grab" icon. Then moving the mouse (while keeping the middle button down) will cause the text object to track the "grab" cursor until you release the middle mouse button. Any screen damage caused by dragging text objects over other objects can be cleaned up with a **RESHOW**.

Text objects can be removed by clicking the right mouse button over them. A warning will then appear on the message panel asking you to confirm the operation by pressing the right mouse button again which then will (permanently) delete the text. Thus you need to click "right" twice over a text object to delete it.

If logging is enabled, all commands that create, delete or move text objects are also written to the log file allowing that text to be recreated and positioned automatically when the log file is read in. In fact, if you wish to change fonts of text objects, one way to do it is to edit the log file items that created the text items with the new font before reading it. (See section: *GI command interface*).

7.5 Draw Mode – line drawings

It may also be useful for you to be able to draw objects on the screen. "Draw" mode (see Figure 4) allows you to do this in one of two ways. You can draw "Line" objects consisting of connected straight line segments or box objects consisting of rectangles. Select which one you want by clicking over the "Lines" or "Boxes" switches of the **TYPE** prompt on the control panel. If you are drawing "Lines", click and release the left mouse button on the display panel where you want to start drawing. This will position a dot (vertex) at the cursor position. Moving the mouse will cause a line segment to emanate from that vertex to the current cursor position. Clicking and releasing the left button again will cause a new vertex to be placed at the cursor and a line segment drawn between the original vertex and the new one. Moving the mouse will then cause a new line segment to emanate from the new vertex to wherever the cursor is. By continuing to move the cursor and pressing the left mouse button you can create a line drawing of almost any complexity. To stop drawing just click left twice in the same place.

Boxes are drawn similarly except that only two diagonal corners of the box need to be specified. Clicking left and releasing will create one corner; moving the mouse will then cause a box to emanate from that corner to the current position of the cursor. Clicking down and releasing the left button will then set the other corner and the box object will have been created.

Boxes and line drawings are treated similarly to text items. You can move them around with the middle mouse button (but you must "grab" them at a vertex) or delete them with the right button (again you must be near a vertex). If logging is enabled, commands that generate, move or delete these drawn objects are written to the log file. GI limits each line object to no more than 10 vertices and will automatically break up objects you try to draw that are larger than this into two or more separate objects. Like text objects, if you exceed this limit, you will ordinarily be unaware of it until you try to move or delete the object(s).

7.6 Custom Mode – customizing mouse buttons

Custom mode is a little more complicated, but if you do a lot of simulation work, learning how to use it may save you a lot of time. When you get into custom mode, the mouse buttons no longer have any pre-specified actions on the display panel. Instead the actions of the mouse buttons are defined by you. The control panel has 6 prompts with little icons in front of them that (are supposed to) look like mice (see Figure 5). The top three are for down button actions (left, middle and right) and the bottom three are for up (release) button actions. If you don't define anything for a button (leave it *null or blank) nothing happens when that button action occurs. However if you do specify a command, then when that button action occurs while the cursor is over the display panel, that command will be executed just as if were typed into the command panel. Thus what custom mode allows you to do is map GI or simulator commands to mouse buttons. For example, if you do a lot of resetting of your network, you can map the simulator command "reset" to the left mouse button. Then every time you press the left mouse button over the display panel, the simulator will do a network reset. The mouse buttons only act that way while you are in custom mode; when you go back to any of the other modes, the mouse buttons revert to their old meanings.

In order to make your customized commands more useful, you are allowed to create commands with symbolic arguments that are filled in at the time the mouse button is pushed and whose values depend on where the mouse is on the display panel. For now there are three substitution arguments you can use: **\$u**, **\$x** and **\$y**. (Future releases may have more, so avoid using "\$" names for anything). They have the following meanings and values:

- **\$u**: returns the unit index of the unit icon (if any) underneath the mouse cursor. If there is no unit icon underneath the cursor the command will not be executed.

- **\$x**: returns the x (horizontal) pixel coordinate of the mouse cursor in **display space**.
- **\$y**: returns the y (vertical) pixel coordinate of the mouse cursor in **display space**.

You can use as many substitution arguments as you wish, but they must each be a separate argument in the command. That is, they must be surrounded by white space. So for example, you could map the left button to the command "pot \$u 1000". Then every time (while in custom mode) you clicked the left mouse button over a unit icon, the potential of that unit would be changed to 1000.

You can specify multiple commands to be executed sequentially with the press of a button by separating the commands by a semi-colon (":") surrounded by blanks on both sides. You can also specify a partial command on one button with the command continued on another by making the last argument on the first part of the command two dashes ("-"). This feature could be useful when you need to pick up substitution arguments from different parts of the screen (since the mouse can only be in one place at a time). For example, if you wanted to make a link between two units shown on the screen you could map the first part of the MakeLink command (which needs the unit index of one unit) to left button down and the latter part of the MakeLink command (which needs the unit index of the other unit) to another button action, say left button up. Then by placing the mouse cursor over a unit, pressing the left button down then moving the cursor to another unit (or even the same one) and letting the button up, a MakeLink command will be executed that makes a link between those two units.

The fact that there doesn't seem to be space for a command longer than a couple of dozen characters for each mouse button is an illusion: there's actually two lines. When you get to the end of the line, your keystrokes will automatically be continued on the second line. When you get to the end of the *second* line, you can still keep typing (for up to about 120 characters) with characters at the front of the second line disappearing as you type characters that appear at the end of the line. Those characters that disappeared are still part of the command, they just don't show up in the prompt. Thus you will only be able to see the prefix and suffix of really long commands. However the entire command will still appear on the command panel when they are executed as well as in the log file if logging is enabled.

While commands that you map to mouse buttons are written to the log file when executed, the mappings themselves do not unless you specifically ask them to be. Once you have set up the buttons as you desire, you can write those mappings to the log file by clicking over the **LOG DEFINITIONS** button beneath the last button prompt. You may, for example, have several different mappings that you use, and want to write them to separate files so that you can set them up by executing a simulator "read" command. Obviously to make effective use of "Custom" mode you will have to become familiar with the actual simulator and GI commands. We assume you are already know the simulator commands; the GI commands are discussed in detail in section 10.

8 The Display Panel

The display panel is, of course, the *raison d'être* for GI. The whole purpose of all the other panels is really just to put this piece of pixel real estate to best use in displaying the salient features of your running network. Although the display panel doesn't have any buttons or prompts per se, there are a number of things you can do using the mouse depending on what mode you are in.

First of all, you may have noticed that the display panel seems kind of puny for displaying more than a few dozen nodes. Never fear, stretching the GI window in the normal Suntool fashion will expand the display panel in both directions to the limit of the screen. (And of course you could of made it larger initially by using the `-Ws` flag). In making it larger, you will notice that any "hidden units" (not a joke) that were outside the display, will appear automatically if you make the display panel large enough to encompass them. They were really there all along, they just were not within the current scope of the display panel. Another way to bring units that are currently outside the panel into view without making the panel larger is to "move" (translate) the panel itself. This is done by pressing the middle mouse button down on a piece of screen that does not have a object under it (otherwise you'll move the object and not the display). You should notice the cursor changing to the "grab" icon again, except in reverse image: this indicates that you have indeed grabbed the screen and not something else. Now (while still holding the middle button down) move the mouse in the direction that you want the display window to move (the display doesn't actually move, of course, but all the objects in it do). You will notice that as you move it, the **Origin** prompt on the control panel will change reflecting where the new origin will be. The display itself will not track the mouse (it would be too slow) but when you release the middle mouse button, the whole display (i.e. the object in it) will suddenly jump to reflect the new origin. GI remembers the amount and direction of the last "jump" and will translate the origin the same amount repeatedly if, while holding the middle button down, you press the click the right mouse button. You can similarly go in the opposite direction by pressing the left button. Moving this way is called "jumping" and is a convenient way to "scroll" through two dimensional display space.

The Origin coordinates always indicate where the current display panel "window" is located in something called "display space". The way to think about the relation between "display space", the display panel and the "Origin" is as follows: Display space is a Cartesian plane stretching (almost) infinitely in both directions. The display panel is always showing a finite portion of this plane, specifically the rectangle whose upper left hand corner is located at the origin coordinates in display space. Initially the Origin is "0 0" meaning the display is looking at the positive quadrant of display space. However through the actions just described, any point in display space can be viewed and the Origin coordinates indicate just where in display space the display panel is currently "looking". Usually you will not have to think much about the actual coordinates since most of the work of laying out and examining your network can be done using the marker and appropriate mouse actions.

What the mouse buttons do on the display panel depends on what mode you have GI in. Although most of the mouse functions have been covered when discussing the different control panel prompts, we'll try to summarize them here by mode. Note that custom mode is absent since you define those button actions yourself.

LEFT BUTTON (mark)

- *Main mode:* If the cursor is over a unit icon, causes the detailed information about that unit to be displayed on the info panel (at the column marked NEXT). If there is no unit there, it moves the marker to the cursor position and redisplay it if it had been made invisible. Everywhere else, it should have no effect.
- *Link mode:* If the cursor is over a unit icon, causes the detailed information about that unit to be displayed on the info panel (as in main mode). Otherwise, does nothing.
- *Text mode:* Marks the start position of a new text string. Text string is terminated by moving cursor outside the window or pressing any other mouse button.
- *Draw mode:* Marks individual vertices for line drawings or marks opposite corners of box drawings.

MIDDLE BUTTON (move)

In all modes (excepting Custom) the middle button is used to move objects around on the display screen or to move the display window itself around in display space. You simply click down over the object (or the display window if no object is underneath) and move the mouse with the button depressed and then releasing the button when the object is positioned as required. "Jump" moves with the window can be made by holding the middle button down and then depressing the right or left buttons.

RIGHT BUTTON (target/erase)

- *Main mode:* If the cursor is over a unit icon, it causes that unit to be reverse imaged (marking it as the current target) and copies its unit index into the "target" field of the control panel WHAT prompt. Clicking over a marked unit, "unmarks" it (although its unit index remains in the target prompt). If the cursor is over the marker, it nondisplays the marker. Otherwise the right button does nothing.
- *Link mode:* If the cursor is over a unit icon, it causes that unit to be reverse imaged and marks it as the current target unit. This causes all other units on the display to immediately change to reflect the value of a link between them and the now marked target unit. Otherwise the button does nothing.
- *Text mode:* If over a text object, attempts to delete that text object but first issues a warning message requiring you to click the right button down again to confirm the delete.
- *Draw mode:* Similar to text mode: if over a vertex of a drawn object, attempts to delete that object after asking for a confirmational right button click.

9 The Info Panel

While the purpose of GI is to give a birds-eye view of the behavior of a large number of units at a time, it is sometimes necessary to focus in on one or a small number of units. That is the purpose of the info panel. It has a number of different columns each capable of displaying detailed textual information for a particular unit. Note that as the GI tool window is stretched horizontally, the number of columns available for this type of information increases up to a maximum of eight. Initially none of the columns displays any information for any unit. The way to "activate" one of the columns is to click left over a particular unit on the display panel in main or link mode (or issue an "info" command - see section 10). The particular info column that will display that unit's information is the one that has the "NEXT" icon reverse-imaged just below it. You can specify which column is to be "next" by clicking left over its NEXT icon. Otherwise the NEXT column will automatically circulate to the right. A column on the info panel that is displaying information for a particular unit will continue to "track" that unit as the simulation proceeds, updating itself just as the display panel does. You can clear the information for a particular column by clicking over its NEXT icon when its NEXT icon is reverse-imaged. (i.e. click twice over its NEXT icon).

There are several things to note about the columnar information that is displayed. First is that most of the time you will notice that one of the values in each column will be bold-faced. This indicates the aspect of the unit that is currently being captured by the display panel if the unit is being displayed. Secondly, if the aspect for that unit is not Link/in or Link/out, then (normally) no values for these items will be shown. However if the unit is displaying link weight or you are in link mode then the "Link:" field will contain a link weight and an additional field will show up underneath the "Link:" item, labeled "Target" which will contain the name of the unit on the "other side" of the link as well as the site name (if specified). In addition, the label "Target" will have either a ">" or a "<" following it indicating if the link being shown is "to" or "from" the target.

10 GI command interface

As has been hinted at above, most everything you can do (with the exception of link mode) with the buttons and mouse actions, can also be done with commands. This section documents what those commands are and defines their syntax. These commands can be entered on the command panel, read in from a file, or even executed from user code. It is these commands that are built and then written to the log file when logging is enabled and allows you to recreate your display screen or simulation session. In fact, although the SHOW, CHANGE and ERASE commands are executed by pressing buttons on the control panel, they actually first create a command string which is executed by the same routine that reads commands from the command panel. (You may have noticed that a "gi" command appears on the command panel when you execute a SHOW, CHANGE or ERASE through the panel buttons). Thus, although you don't need to know the syntax of the commands to use GI, that knowledge is necessary for creating or modifying a command file or for programming the mouse buttons in Custom mode.

All the commands follow the same format: They begin with "gi" (indicating that the command is for GI rather than the simulator) followed by an argument consisting of a single letter that specifies the particular command, followed by an argument list containing some number of positional arguments specific to that command. The arguments must be in the exact order indicated, and at least one blank must separate each argument. An argument having blanks as part of it must be surrounded by double quotes. Double quotes that are meant to be part of an argument which itself is double-quoted, must be doubled. If you are building these commands in a file (to be read in by GI using the "read" command) the commands must be separated by a line feed with exactly one command per line (commands cannot cross line boundaries).

10.1 Placing units on the graphics display

The first three commands discussed are the SHOW, CHANGE and ERASE commands which are grouped together because they partially share the same argument list definition:

SHOW:

```
gi s <who> <num> <what> <lrange> <hrange> <target> <image>
      <xstart> <ystart> [<xspace> <yspace> [<numrow>]]
```

CHANGE:

```
gi c <who> <num> <what> <lrange> <hrange> <target> <image>
```

ERASE:

```
gi e <who> <num>
```

where the parameters in <> have the following syntax:

<who> indicates the starting unit for the command and has the same syntax as the WHO prompt on the control panel: either a unit number, name, type or set.

<num> is a decimal number specifying how many units are affected by the command (beginning with the units specified by <who>).

<what> is a character string indicating the aspect of the unit and must be either "P" (potential), "O" (output), "S" (state), "D" (data), "Li" (Link/in) or "Lo" (Link/out).

<lrage> is a decimal number specifying the lower bound of the range of values the selected aspect of the unit will assume.

<hrange> is a decimal number specifying the upper bound of the range of values the selected aspect of the unit will assume.

<target> specifies the target unit if the aspect is Link/in or Link/out and has the same syntax as <who>. If the aspect is other than Link/in or Link/out, this parameter should be 0. If you wish to specify a site as well, append it to the target unit separated by "/". Thus "Mom/apple_pie" specifies a link to unit "Mom" at site "apple_pie",

<image> is either a decimal number from 1 to 6 indicating which default icon family to use (1=square, 2=circle, 3=triangle, 4=pentagon, 5=diamond and 6=grey-scale), or the file name of your own icon family if you are using custom icons. See section: *Creating and using your own icons* for more information about customized icons.

<xstart> and <ystart> are decimal numbers specifying the position in display space for the first icon if more than one will be displayed by this SHOW command.

<xspace> and <yspace> are decimal numbers specifying how many pixels you want to separate the columns and rows of icons if more than one unit is to be SHOWN. Does not need to be specified if only one unit is being SHOWN. If the spacing is made slightly (less than the width or height of an icon) negative, the icons will end up overlapping each other. Making the spacing more negative (larger than the width or height of the icon), will lay out the icons in right-to-left and/or bottom-to-top order.

<numrow> is a decimal number indicating how many icons you want per row if SHOWing more than one icon. The default "max" indicates that all the icons should be placed on the same row and like <xspace> and <yspace>, does not need to be specified if not more than one unit is being SHOWN.

For example, the command:

```
gi s input all P 1 99 0 3 -100 200 40 50 5
```

would attempt to SHOW the potentials of all remaining unshown "input" units using the triangle icons, with the first icon being displayed at coordinate (-100, 200) with 40 pixels separating each icon horizontally and 50 pixels separating each vertically and 5 icons per row. The expected range the potentials will take is from 1 to 99. Note that even though the target parameter is ignored it still needs to be specified ("0") as a placeholder.

10.2 Drawing lines or boxes and adding text

The next set of commands are for creating and deleting text or drawn objects and moving those objects around on the display:

DRAW:

```
gi d <#vertices> <x1> <y1> <x2> <y2> [<x3> <y3> ... <x10> <y10>]
```

draws an object consisting of connected line segments starting at display coordinate (x1, y1) and ending at (xn, yn) where n is between 2 and 10. The <#vertices> argument is a decimal number indicating the total number of vertices (and should thus be n) and will thus always be one more than the number of line segments drawn. Thus

```
gi d 5 50 100 50 200 150 200 150 100 50 100
```

would draw a box with opposite corners at (50, 100) and (150, 200). Note that 5 vertices were needed to draw the box as a closed figure.

TEXT:

```
gi t <string> <x1> <y1> [<font name>]
```

creates a text string consisting of <string> at display coordinates (x1, y1) using the named font. If <string> contains embedded blanks, then it should be surrounded by double quotes. Double quotes that are part of a quoted string should be doubled. If the font is in the directory /usr/lib/fonts/fixedwidthfonts/ only the file name of the font need be supplied, otherwise it must be fully qualified. If not specified or "default", the workstation default font will be used. Thus

```
gi t 'lucky units' 100 200 screen.i.14
```

will display the string *lucky units* beginning at location 100 200 using a 14 point italic font.

10.3 Moving and deleting objects

The following commands are used to move or delete icons, text, or drawn objects.

MOVE:

```
gi m <x1> <y1> <x2> <y2>
```

moves either an icon, text or drawn object located at display coordinates (x1, y1) to the location (x2, y2). If no object is at (x1, y1) then the command simply does nothing (an error message is sent to the message panel, though). If more than one object is at (x1, y1) then only one of them (indeterminately) will be moved. Note that for drawn objects, the (x1, y1) must be "near" (within 3 pixels) of one of the object's vertices.

DELETE:

```
gi x <x> <y>
```

deletes a drawn or text object at that display coordinate location (x, y) if there is one. If no text or drawn object is at that location, then the command does nothing but put up an error message on the message panel.

10.4 Simulating and updating the graphics display

GO:

```
gi g [<#steps> [<#update\_steps>]]
```

causes the simulator to run the network for <#steps> steps, updating the display screen every <#update_steps>. Thus has the same function and analogous syntax to the "GO" button on the control panel. Thus

```
gi g 20 5
```

will run the network for 20 simulation steps, updating the display after every 5 steps. If <#update_steps> is not supplied, it defaults to 1, as does the <#steps> argument. Setting the <#update_steps> argument greater than 1 will allow the simulation to proceed faster, but the tradeoff of course is that you won't see the unit icons change after every step.

10.5 Redisplaying

RESHOW:

```
gi r [<x> <y>]
```

redisplays the screen "from scratch" so to speak with possibly a different display space origin (given by <x> and <y>). This is equivalent to interactively using the "RESHOW" button on the control panel. It causes GI to erase the screen and completely rebuild it from scratch using the new display space origin coordinates (if given). Any "damage" on the screen should then get cleared up.

10.6 Displaying unit details

INFO:

```
gi i <x> <y> [<col>]
```

displays the detailed information for a unit with an icon at display space location (x, y) on the info panel in the column (1-8) indicated by the <col> argument. If <col> is not specified, the current "next" column as indicated on the info panel is used. Issuing this command is equivalent to clicking the left mouse button over the unit icon while in main or link mode. If currently no icon is at location (x, y) nothing happens except that a warning is issued to the message panel.

10.7 Mapping mouse buttons

SET BUTTON:

```
gi b <#button> <'command\_string'>
```

maps the indicated mouse button to the specified command string for execution in custom mode. (A separate "set button" command is built for each of the six mouse buttons and written to the log file when the "LOG DEFINITIONS" button is selected in custom mode). The mouse buttons are numbered from 1 through 6 with the left, middle and right buttons (down) numbered 1 through 3 respectively, with numbers 4 through 6 assigned to release of the left, middle and right buttons. Example:

gi b 4 ''pot \u 1000 ; gi i \x \y 3''

sets the left mouse button release action to two commands to be issued sequentially: The first command sets the potential of the unit under the cursor to 1000 and the second then displays the info for that unit in the 3rd column on the info panel.

11 Advanced Features

You should now know enough to make effective interactive use of GI. There are some other sophisticated features available, some of them hidden, that may be useful to you. However for now, it's probably better for you to go out and practice what you've learned so far and come back to this section when you're pretty familiar with the basic operation of GI or when your curiosity is just killing you.

11.1 Creating and using your own icons

You may not like some or any of the default icons we've given you to display units of your network. No problem: there is a way to specify your own icons if you're willing to go through a little trouble. What you have to do is create a "family" of icon files (one icon for each subrange you want to be able to distinguish) using *icontool* (or anything that puts the pixel definitions in the same format). Then, on the main mode **WHAT** prompt, select the icon with the "?" in it - this indicates that you wish to supply your own icons. Then put the file name of the icon family into the "Name:" field. When the **SHOW** or **CHANGE** command is executed, it will look for the specified icon files, read them in and use them for displaying those unit icons. There are some restrictions and rules you have to know to successfully use customized icons. First of all the icons within a particular family have to be the same size. For Sun 2.0 to use them, icons must have a width of some multiple of 16 pixels, but can be of any height. (Sun 3.0 may be more flexible). So if you are using *icontool* you can use both the "icon" (64 x 64 pixels) and "cursor" (16 x 16 pixels) type icons. (The default polygonal icons GI supplies are of the "cursor" type). So one thing customizing icons gives you is control over the size of the icons for displaying units in your network. There is also a naming convention for the files you are going to put the various icons into. The file names for a particular icon "family" should be of the format *yournamc.#* where "#" is numbered from 0 through however many icons are in that family.

For example, say you wanted a set of smiley face icons to represent the output of some units and needed to be able to distinguish 4 different levels of output values. You could then use *icontool* to create 5 different icons named, say, *funny.0*, *funny.1*, *funny.2*, *funny.3* and *funny.4*. The reason you need 5 icons to represent 4 ranges is that *funny.1* through *funny.4* will be used to divide up the output range (whatever it is) with *funny.0* used only to indicate a value of exactly 0. Thus if you set the output range to be -30 to 70, *funny.1* would be used for values (-infinity) to (-6), *funny.2* for (-5) to (19), *funny.3* for (20) to (44), and *funny.4* for (45) to (+infinity). However for the "special" value of 0, *funny.0* would be used. If you didn't want this special treatment of the 0 value, you could just make *funny.0* look exactly like *funny.2*. When running GI, select the "?" icon and put the name "funny" into the "Name:" prompt that will then appear. Now doing a **SHOW** or **CHANGE** will result in those "funny" icons being used instead of one of the default polygonal ones. You can have (almost) any number of different icon families at a time (the size of memory is the only limitation). You can specify the same family many different times in a GI session without any additional memory overhead: GI remembers each icon family it loads in and will search through that list first before reading them from the file system. There is also no (practical) limit on the number of icons that can be in a family (ie. the number of subranges). If you want you could specify a separate icon for every discrete value between -1000 and 1000 by creating an icon family (maybe *tribe* would be a better description) with 2001 members.

11.2 The programming interface for GI functions

The normal way one builds a network using the simulator is to create a C function that is callable from the simulator command interface that will build the network, possibly using parameters passed along with the call. From your callable C function you can then call many of the simulator functions directly. Frequent users of the simulator often exploit this to good advantage, for example, by dynamically creating and setting input potentials for a particular network simulation. What makes this technique so useful is

1. the ability to "call" your own C functions from the command interface, and
2. the ability to call the simulator functions from C code.

The point of this section is that you may want to do the same thing with the GI interface. For example you may want to have the job of displaying your network done dynamically by your own C code rather than interactively (which is time-consuming and error-prone) or by reading in a command file (which is inflexible). To allow you to do this, GI provides a programming interface to many of its functions, but in a different fashion from the simulator. Rather than provide and document the calling sequences to actual GI functions, we chose to limit all (documented) programming interfaces to a single routine: `gi_command`. It takes a single argument - a character pointer - and assumes that it points at a command string to be executed. If the command string begins with the argument `gi` then it tries to execute the command itself, otherwise it sends the command to the simulator for execution. Thus `gi_command` treats the command string passed to it as if it had been typed by you on the command panel. It returns to its caller an integer indicating whether the command was successful: 0 indicates (probable) success and -1 indicates (probable) error. In addition, if the command generated any error or confirmational messages, these will be displayed on the message panel. Thus if in your C code you specify:

```
return_code = gi_command('gi d 2 150 200 300 450');
```

executing that code (while in GI) will result in a drawn object being created that consists of a line segment from display space coordinates (150, 200) to (300, 450).

The advantages of this "single interface" approach is that it gives a consistent and easily remembered way to access most GI functions through code. Since `gi_command` is the routine that internally processes all interactive commands, we can guarantee absolute consistency of function and error detection between the interactive and programming interfaces. It also provides "for free" program interfaces to all future GI functions implemented as interactive commands. In addition you only need to remember one syntax for both interactive and program commands. The disadvantage is that it probably takes a little more work to create the command string. However, the syntax of GI commands has purposely been made simple, terse and rigid; given C's powerful string commands (especially `sprintf`) building the appropriate GI command in code should be fairly easy.

11.3 Using the log file

As noted previously, all commands typed into the command panel and executed, as well as most commands performed by mouse actions or panel buttons, are made into command strings and written to the log file (if logging is active). Logging is enabled and disabled by selecting the **LOG:** switch on the mode panel. (See previous section: *The Mode Panel*). As was mentioned there, the reason you might want to do this is to save yourself time and trouble the next time you are running that same network. Especially if you have spent a lot of time getting your network displayed exactly as you want it, you'll appreciate only having to do that once. However you have to be a little knowledgeable regarding the log file if you are to get maximum benefit from it. This section will try to give you some hints on possible log file usage and how, combined with the read command, it can be useful to you.

First of all, you should know exactly what commands GI has the capability of logging in the first place. To begin, basically all commands that affect the appearance of the display panel are logged. Thus all **SHOW**, **ERASE** and **CHANGE** commands are logged. Similarly commands are generated and logged whenever you create, move or delete a text or drawn object. **"GO"** and **"RESHOW"** commands are logged as well, since their execution has the potential to change the display panel. Also an **"info"** command is logged whenever you click left over a unit icon to display its values on the info panel. Finally, any command at all that you type directly into the command panel and execute, whether it be a simulator command, a GI command or a call to your own function, will be logged. This includes commands executed by mouse actions in custom mode, if those mouse actions are mapped to commands. If such commands have substitution parameters, they will be logged after those parameters have been resolved.

You should also know what commands are *not* logged. None of the mouse actions for displaying Links *n* link mode are logged. This was because it was felt that such commands are really only useful interactively - thus no **"show links"** command exists. (This may change at a later release). Also setting up the custom buttons will not automatically write out a **"set button"** command unless you explicitly request it by selecting the **"SET DEFINITIONS"** button. There is also the special case of the **"read"** command. Executing a read command may cause a number of other commands to be executed - namely those in the read file. Although the initial **"read"** command is logged, the commands executed as a result of that read will not be logged. The reason is that, if you think about it, reading in of the subsequently created log file would execute all those commands twice: once when the read command is executed, and then again for each individual command that was in the read file. Thus only the bottom level **"read"** command is logged. Also not logged are commands that simply sets operational characteristics of the GI tool, for example, enabling logging or switching modes.

The obvious way to use a log file created during a previous session, is to use the simulator **"read"** command to read that list of commands in again, thereby **"replaying"** the GI session. Since the log file was a record of anything significant that happened, you should be able to recreate the session (almost) exactly. Since, strictly speaking, the **"read"** command is a simulator command, you may wonder what the simulator is going to do with all the *gi* commands since it doesn't recognize them. The answer is that in reality the simulator doesn't ever see any of the *gi* commands because when GI is active, the **"read"** command is actually processed by GI itself. What GI does is read each command from the read file and if it starts with a *gi* (or if it is another **"read"**) it executes it itself, otherwise it passes it to the simulator for processing. Not only does this solve the problem of recording both simulator and GI commands in the same file, it also will allow you to use your command files stored on the Sun file system when the parallel simulator is running on the Butterfly Multiprocessor.

When the GI tool first comes up, by default logging is always enabled and the default log file is named *gi.log*. The first thing to remember is that anything that was previously in *gi.log* before GI was started, will get cleared out and rewritten after the first command that is logged. So anything that gets saved in *gi.log* in the current session will be erased the next time you run *gi.log* unless you do something. There are several ways to deal with this:

1. Once you've started the GI session, immediately change the name of the log file to something else; this will prevent you from accidentally clobbering it next time, or
2. After finishing your GI session, remember to move or copy the contents of *gi.log* to another file. Obviously solution (1) is a safer but requires some forethought.

The next thing about using the log file is to make sure that what gets put into the log file is exactly what you want. For example, if you want the log file to contain only network set up and display commands, then you certainly don't want "go", "reshow", or simulator commands in the log file when you read it back in. On the other hand, you may want the log file to contain only those kinds of commands, that is, commands that run the simulation rather than set up the display. Or you may just want a log file to contain specific commands, such as ones that set up the custom mode mouse buttons. Again there are several ways to accomplish this. The first is to keep in mind the kind of a log file you wish to create and simply turn logging on or off at the appropriate times. As long as you don't rename the log file, turning it on and off during a single session will not destroy its contents; it will just act like a command accumulator. On the other hand, if you don't want to think about that kind of stuff during the session, you can simply record everything and later on just edit out all the commands you don't want. A third option is to use a file processor like *awk* or *grep* to filter the commands you want from the log file.

12 Multiple unit views

GI believes fairly strongly in the "one man-one vote" principle, paraphrased as "one unit, one view". That is, as you may have noticed, GI doesn't easily let you create more than one icon for a particular network unit. GI tries to prevent this by checking, whenever a SHOW command is issued, if the unit index of the unit you are requesting be SHOWN is already displayed. If it is, GI will normally ignore your request to display another icon for that unit. However there are many situations where having more than one icon per unit could be useful. If you had more than one view you could display several aspects of a single unit simultaneously, for example, its potential, output and state. Or you could show the same unit in different parts of display. The major problem with multiple unit views is finding a good way to specify which one of several that exist when building a CHANGE or ERASE command. We weren't able to come up with a clean way ourselves as a matter of fact, but we did want to allow multiple unit views. Thus the method we've come up with is a bit awkward, but until we come up with something better, here's how it works:

If you already have displayed an icon for a particular unit, and you want to create an additional icon for it, simply prefix the backslash character ("\") to however you designate that unit in the WHO prompt of the SHOW command. When processing the SHOW command, if the first character of the unit designation is a backslash, GI will skip doing the checks it normally does that make sure the unit is not already displayed. It also internally marks the new icon as being an "auxiliary" icon, as distinguished from the "primary" view of that unit that was first created. Thus you can have as many "auxiliary" icons as you want for a particular unit, but only one "primary" icon. The difference between "primary" and "auxiliary" icons will only become noticeable when you do a CHANGE or ERASE command. That is because the backslash prefix is valid on these commands as well; and necessary if you want to change or erase an auxiliary unit. To change or erase a primary icon, you do nothing different since the absence of the backslash in the name will cause GI to look only at the primary icons for that unit. On the other hand, if you wish to change or erase an auxiliary icon, you need to use the backslash prefix in the name so that GI only looks for icons marked as auxiliary. Unfortunately a problem arises if you have more than one auxiliary icon for a unit, and wish to change or erase just one of them. Since GI isn't able to distinguish between more than one auxiliary unit, it will just change or erase the first one it finds on the chain - which may or may not be the one you had in mind. There is currently no good solution to this, except to caution you to set up any multiple auxiliary icons with care, since fixing them later may prove a little frustrating.

13 Performance Hints

Although for small, simple networks performance will probably not be a problem for you, large networks, or those that require many thousands of simulation steps, may tax your patience. Although GI has been designed to be fairly efficient, just the fact that it is writing out graphics commands and queries the status of all the units after every step, means it that it uses a fair amount of machine cycles. So if you running a simulation session with GI where performance is important, there are a few things you can do that will help GI to run faster.

One, of course is not to require GI to update the display after every simulation step. Especially if your network changes rather slowly anyway, you may well be able to get by with only updating the display every 5th or 10th simulation step. You control how often GI updates its display panel via the "update steps:" parameter on the GO command. This will reduce GI overhead to zero between those simulation steps that don't require an update.

Another way to reduce GI overhead is by restricting the display panel to just those units you are interested in watching. Say, for example, you have a network of 2000 units but for the particular simulation, you are only interested in the 200 that make up the "learning" layer. If all 2000 units are on the visible display panel, then every simulation step that requires an update will force GI to do 2000 units of work. On the other hand, by moving the display window or compressing the tool window to focus in on just the 200 of interest, you can cut GI's overhead by 90. This is because internally GI keeps track in separate data structures those units that are currently displayed and those that aren't. Units that are not within the current dimensions of the display window are never even looked at between simulation steps, so "hiding" them when you're not interested in them can result in significant performance improvements.

You can also improve performance through optimizing the number of subranges or, equivalently, the size of the range for the aspect being tracked by an icon family. Since a significant amount of GI overhead results just from having to write a new icon to the display, GI only writes a new icon when absolutely necessary. That is, even if a unit's tracked value changes at a simulation step, if the changed value is not outside the subrange of the currently displayed icon, GI takes care to not superfluously redisplay that same icon. So by coordinating the displayed range with the number of subranges you are actually interested in, you can cut down on the number of times GI has to write out a new icon. For example, let's say you are really only interested in distinguishing when a unit's output is below or above 500; you don't really care if its 250 or 400. If you use the default ranges (-1000, 1000) and the default icons (of which there are 20), then every time the value of such a unit changes by 100, a new icon will have to be displayed. A better strategy would be to change the range to (-5000, 5000) in which case a icon display would be generated only when the unit changes from below 500 to above. Or equivalently, you could design and load in an icon family that only had 3 members - one for 0, one for 0-500 and one for 500-1000 - and then set the range to (0, 1000).

14 Future Directions

This is the first official version of the GI package. We have tried to put in a level of functionality that will satisfy most users most of the time. We expect, of course, that based on user experience there will be requests for more function and performance. We ourselves have identified the following probable areas of enhancement:

- Update to Sun 3.2. The current version of GI was developed on a Sun 2.0 OS. With a recompile it will also run under Sun 3.0 and 3.2 as well. However the graphics package for the Sun was significantly redesigned at release 3.0 (what was SunTool is now SunView) and with it many functional and performance enhancements which could be made use of in GI were it to be rewritten in SunView. Since SunView and SunTool are quite similar (the major difference being in the graphics panel) such a rewrite would probably be worth the effort.
- Provide an efficient Butterfly version of GI. Because of the Butterfly Multiprocessors unique environment, it should be possible to find a very efficient way of interfacing the GI tool running on a Sun with the parallel version of the simulator running on the Butterfly. In fact we have such a design and we expect it to be forthcoming shortly. (Summer of 1987).
- Provide for a customizable "set-up" file for running the simulator and GI. Such a file would make it easy for the user to specify things such as where and how big to make the tool, what kinds of default icons to use, should logging be turned on and what commands should be logged. Basically this would make it even easier for the user to tailor the GI interface to his or her specifications.
- Allow the user to flip back and forth between interfacing to GI and interfacing directly to the simulator (as though GI were not compiled into the simulator object). This would allow the user (and the simulator) to make use of special key sequences that normally would not be translated into appropriate commands by the command panel interface

We hope you enjoy using GI and the simulator. Should you have any problems, complaints or suggestions, please contact the authors through the University of Rochester computer laboratory staff. We will make every effort to make sure GI is a reliable, flexible and most of all, useful tool for your simulation needs

Copy available to DTIC does not
 permit fully legible reproduction

Index:	Index:	Index:	Index:
Name:	Name:	Name:	Name:
Type:	Type:	Type:	Type:
Potential:	Potential:	Potential:	Potential:
Output:	Output:	Output:	Output:
State:	State:	State:	State:
Data:	Data:	Data:	Data:
Link:	Link:	Link:	Link:

NE XT	NE XT	NE XT	NE XT
MODE: Main	Link	Text	Draw
Custom	LOG: On	: gi log	

	WHO Pix
	HOW 20 MANY
	WHAT Potential State Link/in from 0
	Output Data Link/out to 500
	HOW ? name
WHERE start x: 5 y: 5 space x: 20 y: 20 units per row: 5	
<div>SHOW CHANGE ERASE</div> <hr/> Clock= 0 Origin= 0 0	
<div>GO number steps: 1 update steps: 1</div>	
<div>DUMP : gi.image</div>	
<div>RESHOW : 0 0 QUIT</div>	

Change command successful
:: gi c Pix 20 0 0 500 0 1 -> read demo.sq2

Figure 1 (GI Tool Window -- *main* mode)

Copy available to DTIC does not
 permit fully legible reproduction

Copy available to DTIC does not
 permit fully legible reproduction

Index 1188	Index 1189	Index 1190	Index 1191
Name: Vx_3.1	Name: Vx_3.2	Name: Vx_3.3	Name: Vx_3.4
Type: Verter	Type: Verter	Type: Verter	Type: Verter
Potential: 0	Potential: 0	Potential: 0	Potential: 0
Output: 0	Output: 0	Output: 0	Output: 0
State: 0	State: 0	State: 0	State: 0
Data: 0	Data: 0	Data: 0	Data: 0
Link: 1000	Link: 1000	Link: 1000	Link: 0
Target >1196	Target >1196	Target >1196	Target >1196
NE XT	NE XT	NE XT	NE XT

MODE: Main **Link** Text Draw Custom LOG: On : gi.log

TARGET 1196/any

HOW ☒ ☐ ☐ ☐ ☐

from: -1000 to: 1000

DIRECTION **Link/in** Link/out

Clock=0 Origin=278 89

number steps: 1
 update steps: 1

: gi.image

: 278 89

show info | move objects | mark link targ

:: gi e Slope all
 -> read demo.sq

Figure 2 (GI Tool Window -- link mode)

Copy available to DTIC does not
 permit fully legible reproduction

Copy available to DTIC does not
 permit fully legible reproduction

Index: 1188	Index: 1189	Index: 1190	Index: 1191
Name: V_3.1	Name: V_3.2	Name: V_3.3	Name: V_3.4
Type: vertex	Type: vertex	Type: vertex	Type: vertex
Potential: 0	Potential: 0	Potential: 0	Potential: 0
Output: 0	Output: 0	Output: 0	Output: 0
State: 0	State: 0	State: 0	State: 0
Data: 0	Data: 0	Data: 0	Data: 0
Link:	Link:	Link:	Link:

NE XT	NE XT	NE XT	NE XT
----------	----------	----------	----------

MODE: Main Link **Text** Draw Custom LOG: On : gi.log

☐ ☐ ☐ ☐ ☐ ☐
☐ ☐ ☐ ☐ ☐ ☐
☐ ☐ ☐ ☐ ☐ ☐
☐ ☐ ☐ ☐ ☐ ☐

This is the Vertex Array

(attached to Edge Units)

FONT *default

Clock= 0 Origin= 0 0

GO number steps: 1
 update steps: 1

DUMP gi.image

RESHOW 0 0 **QUIT**

mark text start | move objects | delete t

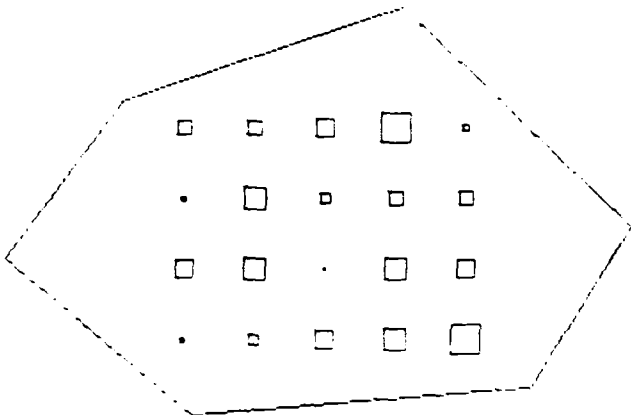
gi e Slope all
 -> read demo.sq

Figure 3 (GI Tool Window -- text mode)

Index: 1188	Index: 1189	Index: 1190	Index: 1191
Name: Vx_3.1	Name: Vx_3.2	Name: Vx_3.3	Name: Vx_3.4
Type: Vertex	Type: Vertex	Type: Vertex	Type: Vertex
Potential: 0	Potential: 0	Potential: 0	Potential: 0
Output: 0	Output: 0	Output: 0	Output: 0
State: 0	State: 0	State: 0	State: 0
Data: 0	Data: 0	Data: 0	Data: 0
Link:	Link:	Link:	Link:

NE
XT
NE
XT
NE
XT
NE
XT

MODE: Main Link Text **Draw** Custom LOG: On : gi.log



This is the Vertex Array
(attached to Edge Units)

TYPE **Lines** Boxes

mark vertex | move object | delete drawing

Clock= 0 Origin= 0 0

GO number steps: 1
 update steps: 1

DUMP : gi.image

RESHOW : 0 0

QUIT

: gi e Slope all
 -> read demo.sq2

Figure 4 (GI Tool Window -- draw mode)

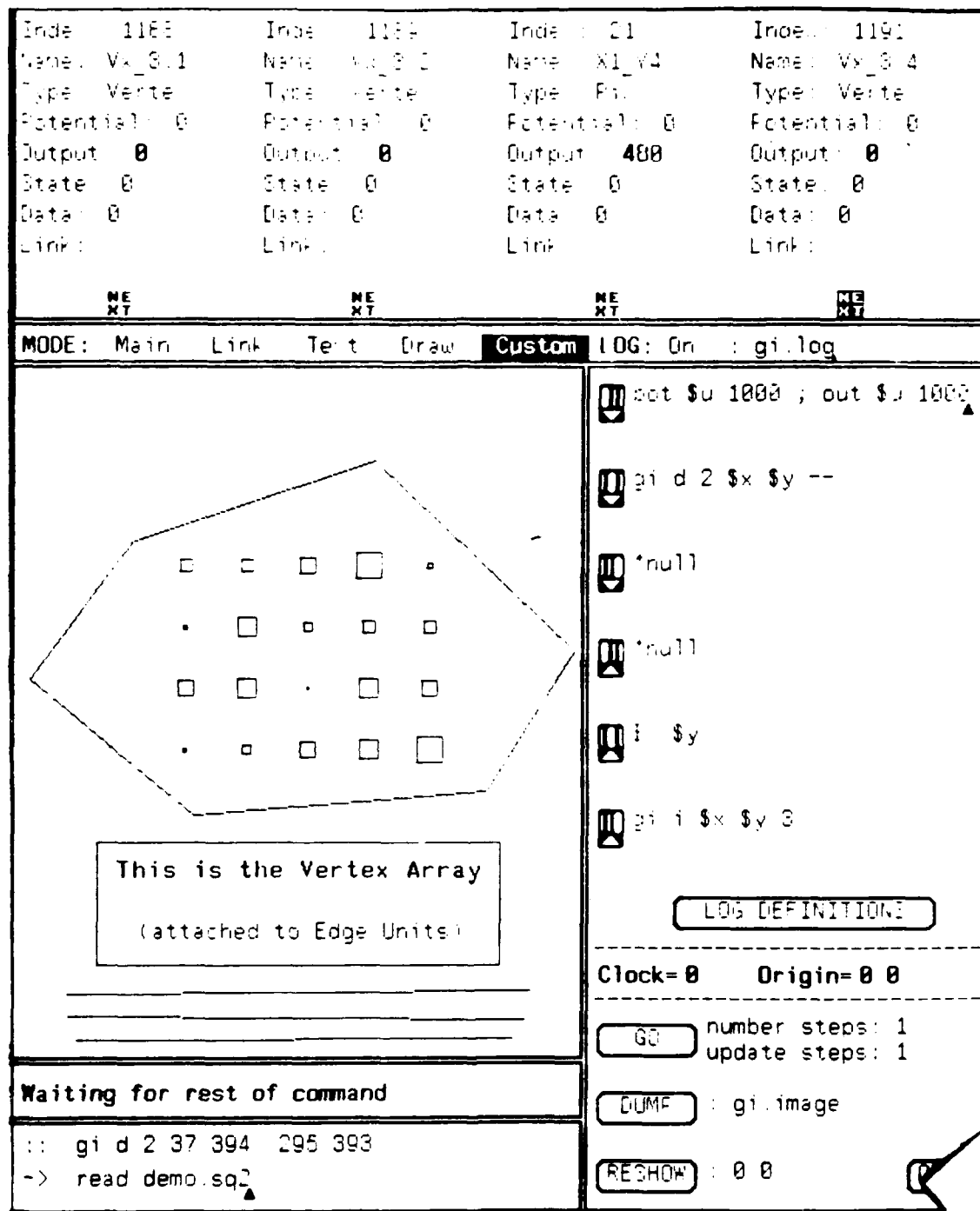


Figure 5 (GI Tool Window -- *custom* mode)

The Rochester Connectionist Simulator
Volume 3:
Advanced Programming Manual

Nigel Goddard
Dept. of Computer Science
University of Rochester
Rochester, NY 14627

April 25 1987

Contents

1	Introduction	3
2	The Network Data Structure	4
2.1	Creating space for units	5
2.2	Making units	5
2.3	Adding sites	5
2.4	Making links	6
3	Naming	7
3.1	Unit names	7
3.2	Function names	7
3.3	Set and State names	8
3.4	Miscellaneous name functions	8
4	Display functions	9
5	Set functions and macros	10
5.1	Adding units to a set	10
5.2	Removing units from a set	10
5.3	Creating and deleting sets	11
5.4	Set theoretic functions	11
5.5	Set membership tests	12
5.6	Miscellaneous set functions	12
6	Modifying and Accessing network values	13
7	Library functions	14
7.1	Unit functions	14
7.2	Site functions	14
7.3	Link functions	14
7.4	Weight scaling	14
8	Unit index and pointer macros	15
9	Flag Macros	16
9.1	Setting a flag	16
9.2	Clearing a flag	16
9.3	Testing a flag	16
9.4	An example	17
10	Simulating	18

11 File functions	19
11.1 Network saving functions	19
11.2 Network checkpointing functions	20
11.3 Logging functions	21
11.4 Other file functions	22
12 Calling the Command Interface	23
12.1 Calling the debug interface	23
12.2 Running another interface on top of the standard one	23
13 Parsing command lines	24
13.1 Lexical functions	24
13.2 Parsing unit specifications	26
13.3 An example of a simulator command function	27
14 Guarded code functions	28
15 Miscellaneous functions	28
16 Simulator Variables	29
17 The Name Table	31
18 Customizing unit, site and link data structures	33
18.1 Redefining the data type	33
18.2 Making a link	33
18.3 The link function	34
18.4 The site function	35
18.5 Linking with simulator code	35
18.6 Displaying, saving, loading, etc	36
18.7 Displaying units	37
18.8 Listing links	37
18.9 Checkpointing and Restoring	38
18.10 Saving and Loading	39
18.11 Unit and Site functions	39
18.12 Completing the example	40
19 Customizing the simulator command interface	41
20 Adding to the Help information	43
21 Avoiding name clashes	43
22 Floating Point version	43
23 Linking user and simulator code	44

1 Introduction

This manual assumes familiarity with the operation of the simulator as described in the *User Manual*. The multitude of simulator functions that can be called from user code are described. Examples of customizing the data structures and command interface are given. The sample networks in the *example* subdirectory use some of the facilities described here. Almost all the functions are used by the simulator itself, so examining the simulator source code will reveal further uses.

2 The Network Data Structure

The network data structure is the heart of the simulator. It is the basis of the representation of units, sites and links. Simply put, it is an array of *Unit* structures, with a linked list of *Site* structures attached to each *Unit* structure, and a linked list of *Link* structures attached to each *Site* structure. These structure definitions are:

```
typedef short weight_type;
typedef int data_type;
typedef int func_type;
typedef short pot_type;
typedef short Output;

typedef func_type (* func_ptr)();
typedef data_type link_data_type;
typedef data_type site_data_type;
typedef data_type unit_data_type;

typedef struct link
{
    func_ptr link_f;           /* link function pointer */
    weight_type weight;        /* weight. can be a float */
    Output * value;            /* can be float, pointer to Outputs array */
    link_data_type data;       /* can be float, or user defined */
    int from_unit;             /* index of unit where link originates */
    struct link *next;         /* next in linked list; NULL if last */
} Link;

typedef struct site
{
    char * name;               /* name of site */
    Output value;              /* can be float; value of site */
    short no_inputs;           /* number of links into site */
    site_data_type data;       /* can be float, or user defined */
    func_ptr site_f;           /* site function pointer */
    Link * inputs;             /* linked list of incoming links */
    struct site *next;         /* next in linked list of sites, NULL if last */
} Site;

typedef struct unit
{
    unsigned int flags;        /* miscellaneous flags */
    char * type;               /* unit type name */
    func_ptr unit_f;           /* unit function pointer */
    char * name;               /* name of unit, or NULL if unnamed */
    pot_type init_potential;    /* can be float; initial potential */
    pot_type potential;        /* can be float; unit potential */
    Output output;             /* can be float; unit output */
    short init_state;          /* unit state after a reset */
    short state;               /* unit state */
    short no_site;             /* number of sites attached to unit */
    unit_data_type data;       /* can be float, or user defined */
    unsigned int sets;         /* set membership bit vector */
    Site * sites;              /* linked list of attached sites */
} Unit;
```

The complex structure is to allow various fields to be either a short integer, as above, or a float for the floating point simulator (see section 22). The data fields may be redefined by the user to extend the network data structure (see section 18).

2.1 Creating space for units

Before any units can be made, the program should specify the total number of units needed. The program may only ask for units once, but need not actually use all the units asked for. The total number of units is specified with a call to `AllocateUnits`, for example:

```
AllocateUnits(100);
```

This allocates data space for the requested number of units. If a program does not explicitly allocate space for units, then by default space for 200 will be allocated.

2.2 Making units

Now units may be made with a call to `MakeUnit`. This function builds a new unit, using space allocated by `AllocateUnits`, for example:

```
int MakeUnit(type, func, init-pot, potential, data, output, init-state, state)
    char *type,
    func_ptr func,
    int istate, state, init-pot, potential, output, data;
```

type is a pointer to a character string, and is simply used for display purposes. *func* is a pointer to the function used to simulate the unit's action. *potential* is the activation level for the unit. *data* is a four byte value for the unit data field described above. *output* is the initial output of the unit. *state* is a short integer representing the initial state value. *init-pot* and *init-state* are the values to set the unit potential and state when the network is reset. `MakeUnit` returns the index in the unit array of the unit created. The first call to `MakeUnit` builds the unit with index 0, and consecutive calls to `MakeUnit` will return consecutive indices. An example of a call to `MakeUnit` would be:

```
unit-index = MakeUnit("retinal", UFsum, 500, 500, 0, 50, 1, 1);
```

The function pointer may be NULL, in which case a function which does nothing will be called by the simulator to simulate the unit action. If not NULL, the function must be either one you have written, or one of the library functions. For simple networks the library functions (see section 7) should be sufficient.

2.3 Adding sites

Once a unit has been created, one or more sites may be attached to it with calls to `AddSite`:

```
Site * AddSite(index, name, function, data)
    int unit, data;
    char *name;
    func_ptr func;
```

index is the index of the unit to which the site is to be attached. *name* is a pointer to a character string which will be the name of the site. *function* is a pointer to the function to be called to simulate the action of the site. *data* is the four byte value to be placed in the site data field described above.

Links to the unit cannot be made until there is a site attached to the unit to which they may go. A call to `AddSite` might look like:

```
AddSite(unit-index, "excite", SFweightedsum, 0);
```

`AddSite` returns a pointer to the newly created site structure. As with units, the function may be NULL, one of your functions, or one of the library functions.

2.4 Making links

A link from a unit to a site on another unit is created with a call to `MakeLink`:

```
Link * MakeLink(source-unit, destination-unit, site-name, weight, data, function)
    int from, to;
    int weight, data;
    char *site;
    func_ptr func;
```

source-unit is the index of the unit where the link originates. *destination-unit* is the index of the unit to which the link is going. *site-name* is a pointer to a character string which is the name of the site on the destination unit at which the link is to arrive. *weight* is the weight to put on the link, and should be within range of a short integer. By convention weights are scaled down by a factor of 1000, thus a specified weight of 500 will be treated as a weight of 0.5. This is to allow weights in the range 0 to 1 without having to use floating point arithmetic. Weights may be negative. `MakeLink` returns a pointer to the link structure created. An example of a call to `MakeLink` might be:

```
MakeLink(unit-index, unit-index, "excite", -500, 0, LFsimple);
```

This would make a link from the unit to itself, to be attached at the site "excite", with a weight of -500 (meaning -0.5), and function `LFsimple`. Such a link could be used to provide exponential decay. As with units, the function may be `NULL`, one of your functions, or one of the library functions.

3 Naming

The Name Table access functions are described in section 17. These functions provide additional possibilities

3.1 Unit names

NameUnit(name,type,index,length,depth)

```
char *name;
int type,index,length,depth;
```

As well as naming a single unit, this function can name a vector or 2-D array of units. The name may then be used during simulation from the command interface, and may also be used during network construction. *name* is a pointer to the character string name to be given. *type* is the type of name - SCALAR, VECTOR, or ARRAY. *index* is the index of the unit to be named, or the first unit in the vector or array. *length* is the number of units if it is a VECTOR, and the number of columns if it is an ARRAY, and is undefined for SCALAR. *depth* is the number of rows for an ARRAY, and is undefined for SCALAR and VECTOR.

```
char * IndToName(u)
int u;
```

Returns a pointer to a volatile string containing name of the unit with index *u*, or **NO NAME if the unit has not been given a name. If the name is that of a VECTOR or ARRAY, the name has the form *name[offset]* or *name[row][column]*.

```
int NameToInd(name, column, row)
char * name;
int column, row;
```

Returns the index of the unit with the given name. If the name is that of a VECTOR, then *column* gives offset of the unit within the vector. If the name is that of an ARRAY, then *column* and *row* give the column and row of the unit within the array. If the name is not that of a unit, or either of the indices are out of range, then the function returns -1.

3.2 Function names

```
char * FuncToName(function_pointer)
func_ptr function_pointer;
```

If *function_pointer* is a pointer to a user function, e.g a unit function, or to a simulator command function, then FuncToName returns a pointer to the name of the function, otherwise NULL.

```
func_ptr NameToFunc(name)
char * name;
```

If *name* is the name of a user function, or a simulator command function, then NameToFunc returns a pointer to the function, otherwise NULL.

```
char * IndToFuncName (index)
int index;
```

Passed *index* into the function table, returns the pointer to that function's name, or NULL if the index is out of range.

3.3 Set and State names

```
DeclareState(name,num)
    char * name;
    int num;
```

Associates a name with a state number. num must be in the range 0 to 99.

```
NameToSet(name)
    char *name;
```

```
NameToState(name)
    char *name;
```

Passed a set/state *name*, returns the set/state number, or -1 if the *name* is not that of a set/state.

```
char *SetToName(number)
    int number;
```

```
char *StateToName(number)
    int number;
```

Passed a set/state *number*, returns a pointer to the set name. If the *number* does not correspond to a set/state, the function returns a pointer to a volatile string containing the character version of the *number*.

3.4 Miscellaneous name functions

```
char * NameToType(name)
    char * name
```

If *name* is the name of a unit type, a unit, a site, a function, a unit state, a set, or an unused name in the name table, NameToType returns a pointer to a volatile string detailing the type of name, e.g. "unit vector name". If the name is not found, it returns a pointer to the volatile string "unknown name".

4 Display functions

ListLinks()

Writes a one-line description of each link to the display output file *DispF* with a header. The line consists of the four values: source index, destination index, weight, and data.

DisplayUnit(u)
int u;

DisplayUnitP(u,up)
int u;
Unit * up;

Writes a complete description of the unit with index *u* to the display output file *DispF*, including descriptions of all the links to that unit. *DisplayUnitP* avoids having to index into the unit array by having the unit pointer passed in as a parameter.

ListUnits(all)
int all;

Used for listing units. If *all* is nonzero (TRUE), it writes a one line description of all units to the display output file *DispF*. If *all* is zero (FALSE) it writes the description only for units with the list flag set. Each line contains the unit index, name ("NO NAME" if not named), type, potential, output and state.

ShowUnits()

Does a show. If a unit has its show flag set, or it has potential greater than the show potential, or it is a member of a show set, then the unit is displayed in detail.

PipeBegin()

If piping is turned on (PipeFlag non-zero) then the display output file *DispF* is set to be a file pointer to the *popen*'ed process whose name is maintained in *PipeCommand*, otherwise *DispF* is set to *stdout*. This function should be called before any of the display functions listed above.

PipeBegin()

The counterpart to *PipeBegin*. If *DispF* is not *stdout* then it closes the pipe and sets *DispF* to *stdout*. This function should be called after the displaying has been done.

LOGfprintf(LOGfprintf(fp,str,arg1,arg2,arg3,arg4,arg5,arg6,arg7,arg8,arg9,arg10)
FILE * fp;
char * str,*arg1,*arg2,*arg3,*arg4,*arg5,*arg6,*arg7,*arg8,*arg9,*arg10;

An augmented-restricted version of *fprintf*. The string *str* is written to file *fp*, with substitution of the *arg*, in order for %s, %d, %f. If *Logging* is TRUE, the string is written to the Log file as well as to *fp*. If *fp* is *stderr* then a message count for the *Graphics Interface* is incremented. If *Format* is TRUE, the function formats the string into lines no longer than 75 characters, and indents all the lines thus formed by 3 spaces.

5 Set functions and macros

5.1 Adding units to a set

```
AddToSet(name, ulow, uhigh)
    char *name;
    int ulow, uhigh;
```

```
AddSet(name, uindex)
    char * name;
    int uindex;
```

Adds units with index *ulow* to index *uhigh* to the set *name*. Returns TRUE if successful, FALSE otherwise. *AddSet* simply calls *AddToSet* with *ulow* = *uhigh* = *uindex*. Does a name lookup to find the set index.

```
AddSetI(setindex, uindex)*
    int setindex, uindex;
```

Macro which adds unit *uindex* to set with index *setindex*. Faster because no name look up. *uindex* must be a valid unit index. For tight loops.

```
AddSetP(setindex, up)*
    int setindex;
    Unit * up;
```

Macro which adds unit pointed to by *up* to set with index *setindex*. Fastest because no name look up and no need to index into unit array. For tight loops.

5.2 Removing units from a set

```
RemFromSet(name, ulow, uhigh)
    char *name;
    int ulow, uhigh;
```

```
RemSet(name, uindex)
    char * name;
    int uindex;
```

Removes units with index *ulow* to index *uhigh* from the set *name*. Returns TRUE if successful, FALSE otherwise. *RemSet* simply calls *RemFromSet* with *ulow* = *uhigh* = *uindex*.

```
RemSetI(setindex, uindex)*
    int setindex, uindex;
```

Macro which removes unit *uindex* from set with index *setindex*. Faster because no name look up. *uindex* must be a valid unit index. For tight loops.

```
RemSetP(setindex, up)*
    int setindex;
    Unit * up;
```

Macro which removes unit pointed to by *up* from the set with index *setindex*. Fastest because no name look up and no need to index into unit array. For tight loops.

NO W196 981

ROCHESTER CONNECTIONIST SIMULATOR VOLUME 1 USER MANUAL

272

(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE

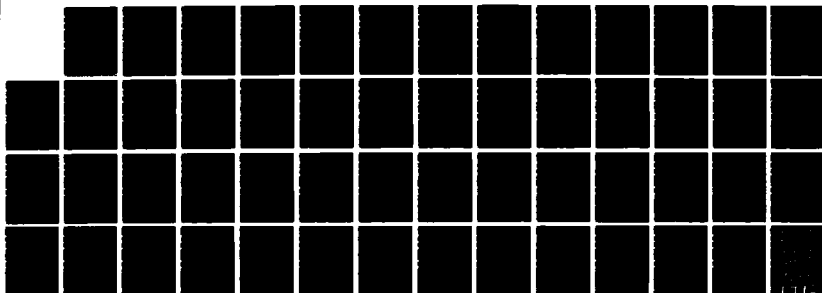
N H GODDARD ET AL 25 APR 87 TR-233-VOL-1

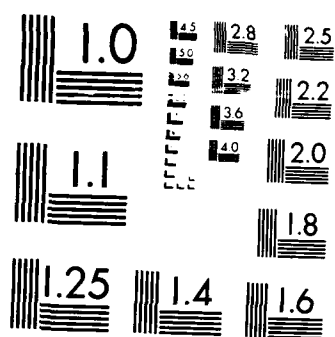
UNCLASSIFIED

N88014-84-K-0655

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

5.3 Creating and deleting sets

DeclareSet(name)
char *name;

Creates a set with the given *name*. Returns the index assigned to the set, or -1 if the set could not be created.

DeleteSet(name)
char *name;

Deletes the set with the given *name*. Returns TRUE if successful. FALSE if the *name* is not that of a set.

5.4 Set theoretic functions

DifferenceSet(name,name1,name2)
char *name,*name1,*name2;

IntersectSet(name,name1,name2)
char *name,*name1,*name2;

UnionSet(name,name1,name2)
char *name,*name1,*name2;

Assigns the difference/intersection/union of set *name1* and *name2* to set *name*. All three sets must already exist. Returns TRUE if successful. FALSE otherwise.

InverseSet(name,name1)
char *name,*name1;

Assigns the inverse of set *name1* to the set *name*. Both sets must already exist. Returns TRUE if successful. FALSE otherwise.

5.5 Set membership tests

```
MemberSet(name,uindex)
    char *name;
    int uindex;
```

Returns TRUE if unit with index *uindex* is in the set *name*. Otherwise returns FALSE, indicating the index was not legal, the name was not that of a set, or the unit wasn't in the set.

```
MemberSetI(setindex,uindex)*
    int setindex,uindex;
```

Macro which calculates TRUE if unit *uindex* is in the set with index *setindex*. FALSE otherwise. Faster because no name look up. *uindex* must be a valid unit index. For tight loops.

```
MemberSetP(setindex,up)*
    int setindex;
    Unit * up;
```

Macro which calculates TRUE if unit pointed to by *up* is in the set with index *setindex*. FALSE otherwise. Even faster because no name look up and no need to index into unit array. For tight loops.

```
MemberSetS(setindex,unitsetbits)*
    int setindex,unitsetbits
```

Macro which calculates TRUE if unit set bits field *unitsetbits* has the *setindex* bit set. FALSE otherwise. Fastest because no name look up, no indexing into unit array, and no adding in offset for unit set bits field. Not much use unless each unit is being tested for membership of several sets.

5.6 Miscellaneous set functions

```
IsSet(name)
    char *name;
```

Returns TRUE if the *name* is that of a set. FALSE otherwise.

6 Modifying and Accessing network values

```
SetOutput(index, value)
SetPotential(index, value)
SetState(index, value)
SetData(index, value)
    int index, value;
```

Unit *index* is given output, potential, state, or data *value*.

```
int GetOutput(index)
int GetPotential(index)
int GetState(index)
int GetData(index)
    int index;
```

Output, potential, state or data of unit *index* is returned.

```
SetWeight(ul,uh,us,usind,ulow,uhigh,uset,usetind,sitename,randomval,val,pert)
    int ul,uh,us,usind,ulow,uhigh,uset,usetind,randomval;
    FLINT val,pert;
    char * sitename;
```

This function sets the weights on one or more links. The first four parameters specify the units from which the links originate. The second four parameters specify the destination units *sitename* is the name of the site at which the links arrive, or optionally ALL, meaning any site on a destination unit. The last three values specify the weight. If *randomval* is FALSE, the weight is simply *val*. If *randomval* is TRUE, then the weights are randomly distributed in the range *val-pert* to *val+pert*.

The source and destination units are in the range *ul/ow* to *uh/igh*. If *uset* is FALSE, all the units in these two ranges are considered. If *uset* is TRUE, then only those units in the range that are in the set whose number is *usind* (for source units) or *usetind* (for destination units) are considered. Thus,

```
SetWeight(0,100,FALSE,0,0,1000,TRUE,6,"excite",FALSE,500);
```

would set the weight on any link from a unit in the range 0 to 100 to a site "excite" on a unit in the range 0 to 1000 which is in set number six to 500.

```
RandomiseWeights(mean,pert)
    FLINT mean;
    int pert;
```

This function sets all the network weights to values in the range *mean-pert* to *mean+pert*. The values are evenly distributed throughout this range.

7 Library functions

There is one library function that is not a unit, site or link function. The other library functions are described in the *User Manual*, but included here for completeness.

```
SiteValue(name, sp)
    char * name;
    Site * sp;
```

If *sp* is a pointer to a linked list of site structures, such as in the *sites* field of the *Unit* structure, and *name* is the name of one of the sites in the linked list, then the function returns the *value* of that site. If no such site is found, 0 is returned and an error message printed.

7.1 Unit functions

UFsum is a unit function which sets output and potential to the sum of all site values.

7.2 Site functions

SFmax sets the site value to the maximum input value.

SFmin sets the site value to the minimum input value.

SFsum sets the site value to the sum of the input values.

SFweightedmax sets the site value to the maximum weighted input value. A weight of 1000 is treated as unity: the input value is multiplied by its weight and the result divided by 1000.

SFweightedmin(up,sp) sets the site value to the minimum weighted input value. A weight of 1000 is treated as unity.

SFweightedsum sets the site value to the sum of the weighted input values. A weight of 1000 is treated as unity.

SFand returns 1 if all its inputs are positive, otherwise 0.

SFxor(up,sp) returns 1 if exactly one of its inputs is nonzero, otherwise 0.

SFprod returns product of inputs.

7.3 Link functions

LFsimple sets the data field of the link to be the input value (unweighted). This does not affect the behavior of the network, but does help with debugging.

7.4 Weight scaling

The library functions assume that weights are scaled up by a factor of 1000. Thus a weight value of 500 represents a real weight of 0.5. This is to allow representation of values in the range 0 to 1 without having to use floating point arithmetic. The floating point library functions also scale by 1000 for compatibility.

Since all the functions that use weights (mainly site and link functions) can be written by the user, any weight scaling factor may be used. The only restriction is that if library functions that deal with weights (such as *SFweightedsum*) are used, weights **must** be scaled by 1000.

8 Unit index and pointer macros

LegalUnit(index)*
int count;

computes TRUE if *index* is the index of an existing unit, FALSE otherwise.

UnitIndex(up)*
Unit * up;

computes the index of the unit pointed to by *up*. If *up* does not point to a unit, computes garbage.

9 Flag Macros

Various macros are defined to set, clear, and test the flags in a unit. Each unit has 32 flags associated with it. Currently flags 0 to 6 are used by the simulator, and flags 7 to 11 are reserved for future simulator use. Flags 12 to 19 should be used for library packages, and so user code should be restricted to flags 20 through 31, preferably working from 31 down. Some of the simulator reserved flags may be set by the user for one or more units.

The user-settable flags are as follows:

SHOW_FLAG	if set then the unit is in the Show set (for the <i>show</i> command).
LIST_FLAG	if set then the unit is in the List set (for the <i>list</i> command).
NO_LINK_FUNC_FLAG	if set then no functions are called for the links into a unit. This will result in speed up.
NO_SITE_FUNC_FLAG	if set then no site or link functions are called for the unit.
NO_UNIT_FUNC_FLAG	if set then no unit, site or link functions are called for the unit. The output of the unit remains the same.

9.1 Setting a flag

```
SetFlag(uindex, flagno)*  
SetFlagP(up, flagno)*
```

SetFlag sets flag *flagno* in the unit with index *uindex*. *SetFlagP* does the same thing, but *up* is a pointer to the unit, thus avoiding indexing into the unit array.

9.2 Clearing a flag

```
UnsetFlag(uindex, flagno)*  
UnsetFlagP(up, flag)*
```

UnsetFlag clears flag *flagno* in the unit with index *uindex*. *UnsetFlagP* does the same thing, but *up* is a pointer to the unit, thus avoiding indexing into the unit array.

9.3 Testing a flag

```
TestFlag(uindex, flagno)*  
TestFlagP(up, flagno)*  
TestFlagF(uf, flagno)*
```

TestFlag calculates TRUE if flag *flagno* in the unit with index *uindex* is set. FALSE otherwise. *TestFlagP* does the same thing, but *up* is a pointer to the unit, thus avoiding indexing into the unit array. *TestFlagF* uses the bit vector *uf* (an unsigned int) and thus avoids adding in the offset for the *flag* field in the unit. *TestFlagF* is only of any use if several flags are being tested for each of many units in a tight loop.

9.4 An example

If flags are used a lot, it is advantageous to be able to use the macros which take a Unit pointer inside loops. An example of how this is done is a code fragment from the simulator source.

```
register Unit * up;
register int which, ucount;

for ( which = 0, up = UnitList, ucount = NoUnits;
      which < ucount;
      which++, up++)
    UnsetFlagP(up, STEP_SIM_FLAG);
```

This code fragment clears the *STEP_SIM_FLAG* for every unit. To avoid having to index into the unit array, *UnitList*, for every unit, the loop maintains a current index, *which*, and a current unit pointer, *up*, both of which are incremented each time round the loop. Now the the pointer version of the flag clearing macro, *UnsetFlagP* can be used, so that no indexing into the unit array is ever done. For an array of thousands of units this can be significant, especially if the code were in a unit function.

10 Simulating

Reset()

Resets the network: sets the system Clock to zero; sets the potential and state of each unit to *init_potential* and *init_state* respectively; sets the output of each unit to zero

Step(count)

int count;

Simulates *count* steps. Echoes and shows will be done if appropriate.

Sync()

Further simulation steps will be synchronous.

Async(seed)

int seed;

Further simulation steps will be asynchronous, as in the *async* command. The random number generator will be seeded with *seed* unless it is zero, in which case it will be seeded with the UNIX system time.

11 File functions

A number of functions are available that deal with files.

11.1 Network saving functions

```
FILE * GetNetFile(fname)
    char * fname;
```

If *fname* is NULL, asks the user for a filename to use for *saving*, opens the file and returns a descriptor to it. If *fname* is not NULL, a file with this name is opened and the descriptor to it returned. Will not overwrite or append to a file without user confirmation.

```
CloseNetFile()
```

Closes the *save* file opened with *GetNetFile* (if one was).

```
NetSave(savef)
    FILE *savef;
```

Writes a time stamp into the file *savef* using *StampTime*, followed by the *structure* of the network (that is enough information to be able to reconstruct all the units, sites and links, including unit names and types and all function pointers. Finally calls *SaveState* to save the *state* of the network (i.e. the weights, outputs, potentials, etc).

```
NetLoad(nfp)
    FILE *nfp;
```

The function complimentary to *NetSave*. Reads in and checks the time stamp in the file *nfp* with *CheckStamp*, reconstructs the network from the file data, and then calls *RestoreState* to restore the *state* of the network.

11.2 Network checkpointing functions

FILE * GetChkFile(fname)
char * fname;

If *fname* is NULL, asks the user for a filename to use for *checkpointing*, opens the file and returns a descriptor to it. If *fname* is not NULL, a file with this name is opened and the descriptor to it returned. Will not overwrite or append to a file without user confirmation.

CloseChkFile()

Closes the *checkpoint* file opened with *GetChkFile* (if one was).

NetCheckpoint(savef)
FILE *savef;

Writes a time stamp into the file *savef* using *StampTime*, followed by the *state* of the network, that is all the weights, potentials, outputs etc. Uses the following function to save the state.

SaveState(savef)
FILE *savef;

Writes the *state* of the network to file for reading in later.

RestoreNetwork(nfp)
FILE *nfp;

The function complimentary to *NetCheckpoint*. Reads in and checks the time stamp in the file *nfp* with *CheckStamp*, and then restores the *state* of the network using the following function.

RestoreState(nfp)
FILE *nfp;

Restores the *state* of the network from file *nfp*.

11.3 Logging functions

FILE • GetCmdFile()

Opens a file for logging the keyboard input only, and returns a descriptor to it. The file name is of the form *run????cmd.#* where *????* is the process ID of the current process, and *#* is an integer.

SaveCmdFile()

Closes the file opened with *GetCmdFile*, and asks the user if it should be saved. If the answer is yes, prompts for a name for the file and does a UNIX environment call to *mv* it to that name. Otherwise it does a UNIX environment call to *rm* the file.

FILE • GetLogFile(fname)
char • fname;

If *fname* is NULL, asks the user for a filename to use for *logging* all i/o, opens the file and returns a descriptor to it. If *fname* is not NULL, a file with this name is opened and the descriptor to it returned. Will not overwrite or append to a file without user confirmation.

AskLogOn ()

Asks the user if (s)he would like to commence a *logging* the i/o. If the answer is yes, calls *LogOn* to open a log file

AskLogOff ()

Asks the user if (s)he would like to close the current log file. If the answer is yes, calls *LogOff* to close the log file.

LogOn()

Asks the user for a file name to use for the log file (supplying a default), opens the file and stores the descriptor to it in *LogFile*.

LogOff()

Closes the file opened in *LogOn (LogFile)*.

11.4 Other file functions

StampFile(nfp,type)
FILE *nfp;
int type;

Writes a time, process and image name stamp to the file *nfp*. If *type* is TRUE (i.e. a *checkpoint* file) the process ID number is written first, otherwise (i.e. a *save* file) the process ID number is not written. Then the function writes out the name of the program that is running (i.e. the simulator executable) and the time it was made in human-readable form. Next it writes out the current time in human-readable form. Finally it writes out the current system time in seconds. Returns -1 on failure. For example:

```
Processid = 882
Image = sim written Wed Apr 29 15:03:58 1987
Current Time = Thu Apr 30 23:32:58 1987
546838378
```

CheckStamp(nfp,type)
FILE *nfp;
int type;

Reads in and checks a stamp made by *StampFile*. If *type* is TRUE, it expects a process ID number, if FALSE it expects no process ID number. The function will return -1 if the check fails because the stamp format is incorrect or missing. It will issue warnings if the file is over a week old, if the stamped image name is different to the current program file name (i.e. the simulator has been recreated), or *type* is TRUE, meaning a *checkpoint* file and the stamped process ID number is different to the current process ID number.

ReadCmdFile(fname)
char * fname;

Open the file named *fname* to read commands from. This may be nested (i.e. a command in one file causing this function to be called to open another command file) to a depth of 16 files.

CloseCmdFile()

Close the file which is currently being read from, and continue reading from the one that caused this function call, or *stdin* if it occurred by the user issuing the *read* command.

12 Calling the Command Interface

It is possible to call the simulator debug command interface from user code, thus allowing a unit function to interrupt a simulation step or other function so that the user can examine or even modify the network mid-step. In fact this is the way the construction debugging facility operates. A more useful feature is the ability to build an interface on top of the simulator interface, which is exactly what the *Graphics Interface* is.

12.1 Calling the debug interface

```
debug_command_reader(str)
    char * str;
```

This function runs the debug command interface. The parameter is a string to be used in the prompt. *Debug* should probably be incremented before calling this function (the value of *Debug* is printed in the prompt) so that it is clear from the prompt how many layers of interfaces are running. For instance, the control.C interrupt routine executes the following code fragment:

```
Debug++;
debug_command_reader('interrupt');
Debug--;
```

resulting in the prompt "interrupt 2>" where *Debug* has value 2. Most of the regular simulator commands can be called from the debug interface, except those to actually run the network. In addition the user can add their own command functions, as described in section 19. The *debug_command_reader* returns when the *quit* command is issued. Normally the user should zero *Errors* before calling *debug_command_reader*.

12.2 Running another interface on top of the standard one

```
char * extern_command_reader(cmd_line)          /* called externally */
    char * cmd_line,                             /* command string */
```

This is the function called by the *Graphics Interface* to pass a command to the simulator. *cmd_line* is a character string containing the command to be executed. *extern_command_reader* executes the command and returns a message string if any output to *stderr* occurred via *LOGsprintf* during the command, or NULL if there was no such output. An interface can be built on top of the simulator, while retaining all the simulator commands, by simply passing on all simulator commands to the simulator via this function. This command may also be used to execute a single simulator command, for example:

```
extern_command_reader('d u 3')
```

would cause unit 3 to be displayed in standard fashion.

13 Parsing command lines

Several functions make it easier to write new simulator commands. These functions will perform some simple lexical analysis and will process command line specification of units, if it is done in the standard fashion. The simulator passes command lines to command functions already parsed into an `argv-argc` structure. Whitespace in the command line indicates argument termination. To use the functions described in this section, the file "lex.h" should be included.

13.1 Lexical functions

```
int Lex(cmd)
    char *cmd;
```

Lex is the main lexical analysing routine used to parse simulator commands. *cmd* is not the whole command, rather it is one argument from the `argv-argc` structure representing the command. *Lex* parses the argument, which should be an integer, a floating point number, a quoted string, a character string containing no whitespace, the character "?", "+", or "-", or a unit identifier of the form *name[index]/[index]*, where both indices are optional. Parsing unit identifiers should be done with *GetUnits* described below.

Lex will return the following values (#define 'd in the standard include file):

Return value	character string
CBRACK]
OBRACK	[
END_STRING	'\0'
ALL	all
AUTO	auto
LINK	c or connections or link or links
CLOCK	clock
DEFAULT	def or default
FUNC	func
FROM	from
IPOT	ipot
ISTATE	istate
NAME	name
ON	on
OFF	off
OUT	out
POT	pot
RANDOM	random
SHOW	sh or show
STATE	s or state or states
SET	set
SITE	site
TYPE	type
TO	to
UNIT	u or unit or units
WEIGHT	weight
HELP	?
STRING	"<characters>"

PLUS	+
MINUS	-
INT	<digits>
FLOAT	<digits.digits>
IDENT	anything other than the above

The most recently parsed token is held in the character string *Yyarg*. The most recently parsed integer is held in *Yyint*. The most recently parsed floating point number is held in *Yyfloat*. Thus command functions can check the return type and get the value or string from one of these three variables, if it is needed.

Ccmdindex is the index into the argument string *cmd* that *Lex* has reached. It is reset to zero when the end of the string is reached. In most cases this will have happened by the time *Lex* returns, but when parsing, say, "RETINA[3][4]", it will return on encountering the first "[", and the next call to *Lex* will continue where it left off. As far as *Lex* is concerned, token delimiters are '\0', '[' and ']'. Unit names, such as "RETINA[3][4]", should be parsed with *GetUnits*.

Curarg is the index into the command *argv* vector indicating the current command argument being processed. It is incremented by *Lex* on encountering the end of the string (*cmd*).

```
UnLex(tok)
    int tok;
```

Unlex effectively puts the most recent token processed by *Lex* back in the input stream, to be re-processed on the next call to *Lex*. Only one token can be *Unlex*'d.

EAT=

EAT is a macro that chews up the rest of the argument currently being processed. It is usually only used if *Lex* returns an unexpected type of token, and command parsing has to be abandoned. *EAT* then resets *Ccmdindex* to an appropriate initial value for processing the next command. See section 19 for an example of the use of *Lex*, *EAT* and *GetUnits*.

13.2 Parsing unit specifications

```
GetUnits(argc,argv)
    int argc;
    char ** argv;
```

GetUnits processes a specification of a range of units, given in the normal form (see the *User Manual* for details). It sets the variables *Ulow*, *Uhigh*, *Uset*, and *Usetind* to indicate which units were specified. *Ulow* and *Uhigh* indicate the low and high unit indices in the range: if they are equal, a single unit was specified. *Uset* is a boolean indicating whether a set of units was specified, and if it is TRUE, then *Usetind* holds the number associated with the set (i.e. the index into the sets bit vector in the *Unit* structure).

GetUnits returns FALSE (=0) if it fails, that is if a valid unit specification was not found. Otherwise it returns one of the following values:

Return value	meaning
FALSE	invalid unit specification
ALL	all units
TRUE	other valid unit specification

```
FOR_UNITS(index)*
FOR_UNITS(index,unitpointer)*
```

FOR_UNITS is a macro to use with unit range specifications. The parameter is an integer, which is used as a unit index inside a *for* loop to cycle through the unit range, using the values set by *GetUnits* in *Ulow*, *Uhigh*, *Uset*, and *Usetind*. For instance, the code:

```
FOR_UNITS(z)
    SetOutput(z,10);
```

would cause all the units in the specification parsed by *GetUnits* to have their outputs set to 10. An example of the use of *FOR_UNITS* in conjunction with *GetUnits* is given in the next section. *FOR_UNITS_P* is simply a version of *FOR_UNITS* that uses a unit pointer rather than a unit index, so that the code in the *for* loop need not index into the unit array. *FOR_UNITS_P* does the indexing itself in a fast manner, so this is the preferred macro for speed.

13.3 An example of a simulator command function

As an example, here is the code for the disp command.

```
Cmd_disp(argc,argv)
    int argc;
    char ** argv;

{
    register int u;
    register Unit * up;

    if ((argc == 2) && (Lex(argv[1]) == HELP))
        goto helpinfo;
    if (argc > 2)
    {
        Curarg=1;                                /* first command argument */
        if(Lex(argv[Curarg]) == UNIT)
        {
            if(!GetUnits(argc,argv)) return 0;
            PipeBegin();
            FOR_UNITS_P(u,up)
                DisplayUnitP(u,up);
            PipeEnd();
            return 0;
        }
        else
        {
            EAT;
            goto synerror;
        }
    }
    else
        goto synerror;

helpinfo:
    Format = TRUE;
    LOGfprintf(DispF, "The d(isp) command is used to display the
values associated with one or more units, for instance the potential,
output, state, functions, site names and values, link weights and values.\n");
    Format = FALSE;
synerror:
    LOGfprintf(DispF, "\nUsage: d(isp) u <UnitID>\n\n");
    return 0;
}
```

As described in section 19, this function obeys the standard format for command functions. First it tests to see if help information has been requested, and if so jumps to the *helpinfo* label. Then it sets *Curarg* to 1 (*argv[0]* is the command name) and calls *Lex* to parse the expected string "units", and tests that it was indeed found. The call to *Lex* will have incremented *Curarg*, so now *GetUnits* is called to find the units which should be displayed. If further parsing were required, *Curarg* would have the correct value, but we expect no more arguments for this command so proceed to display the units. *PipeBegin* is called to set up the pipe process for displaying (if one has been requested with the *pipe* command), and then the *FOR_UNITS_P* is used to call *DisplayUnitP* for all the units in the range.

14 Guarded code functions

When a control_C interrupt is issued from the keyboard, the simulator enters the *interrupt interface* (see section 12.1), so that the network may be examined and possibly modified. To guard against entry to this interface when the network data structure is in the middle of being modified (and may therefore have inconsistent or invalid pointers), functions are provided to delay entry until a safe state is achieved.

Guard()
Release()

Guard simply increments *Guarded* and exits. *Release* decrements *Guarded* and checks if an interrupt has occurred since the previous call to *Guard*. If so, and if *Guarded* is zero, the interrupt processing routine is called to enter the interrupt interface. The fact that *Guarded* is incremented and decremented by each matched pair of calls to *Guard* and *Release* means that these calls may be nested, and interrupts will only be processed when the outermost level is reached.

15 Miscellaneous functions

NullFunc()

This is the function that does nothing. It is used for units, sites and links whose function specification is NULL in the call to *MakeUnit*, *AddSite* or *MakeLink*.

UserWait(str)
char * str;

Prints the provided prompt, *str*, then reads a single character in cbreak mode. Returns the character, mapped to lower case. Used to pause during simulation or display.

16 Simulator Variables

Many variables used by the simulator are accessible to user code. Modifying these should be done with care. The complete list follows.

int AutoFix A boolean value that indicates whether automatic correction of construction errors is enabled.

int Clock The system clock, or count of simulation steps.

FILE * CmdFile The file to which keyboard input is written, if *LogCmd* is TRUE. The file is closed on exit from the simulator, and saved at user discretion.

int Cmdindex Index into the current character of the current command argument. Used by *Lex*, see section 13.1.

int Curarg Index into *argv* structure for command functions, indicating next argument to process. See section 19.

int Debug Indicates the debug and interface level. Zero means normal interface, debugging switched off. One means normal level, debugging switched on. Incremented during construction commands such as *MakeUnit*.

FILE * DispF The file to which display output is written, i.e. during a *display*, *list*, or *help* command. If piping is enabled (see section 4), the pointer is to the opened process. Otherwise it points to *stdout*.

int Echo A boolean value that indicates whether echoing is enabled.

int EchoStep Number of steps between simulator echo messages.

unsigned int Errors A bit vector containing the error types when debugging.

int ExecFract During fair asynchronous simulation (*fsync* command), the percentage of units simulated each step.

int ExecLimit During fair asynchronous simulation, the number of steps before all units have been simulated at least once.

int Format A boolean value indicating whether *LOGprintf* (see section 4) should format the output.

int Guarded Incremented by the *Guard*, decremented by the *Release* function (see section 14), when zero code is not guarded against interrupt processing.

int LastSet Maximum number of sets allowed (currently 32).

int LastUnit Index of last unit that there is space for. Set in *AllocateUnits*

int LogCmd A boolean indicating whether keyboard input is being saved in the command log file.

FILE * LogFile The file to which the log is written, that is the record of all keyboard input and simulator output.

int Logging A boolean indicating whether logging is enabled.

int NoLinks Actual number of links made so far.

int NoSets Actual number of sets used currently.

int NoStates Maximum allowed number of states with names.

int NoUnits Actual number of units made so far.

Output * Outputs Vector of unit output values. Links get their values with a pointer into this array. Updated in *Step*.

int Pause A boolean indicating if *pausing* is enabled.

char PipeCommand [] Name of pipe process to use for display output, if *pipng* is enabled.
int PipeFlag A boolean indicating whether *pipng* is enabled.
char **SetNames Array of pointers to set names, indexed by set number.
int Show A boolean indicating whether *showing* is enabled.
unsigned int ShowSets A bit vector indicating which sets are in the Show set.
int ShowPot During a *show*, display all units with potential higher than this value.
int ShowStep Perform a *show* every this many steps.
int StateCount Actual number of states with names.
char ** StateNames Array of pointers to state names, indexed by state value.
int SyncFlag Simulation update protocol. Can be SYNC (synchronous, *sync* command), FAIRASYNC (fair asynchronous, *fsync* command) or ASYNC (asynchronous, *async* command).
int Uhigh, Ulow Set by *GetUnit(s)* (see section 13.1), indicates range of units found.
Unit * UnitList Pointer to unit array, the main data structure.
int Uset, Usetind Set by *GetUnit(s)* (see section 13.1). *Uset* is a boolean indicating if a set name was found, in which case *Usetind* is the set index.
char Yyarg[] Contains the token most recently parsed by *Lex*.
float Yyfloat Contains the floating point value most recently parsed by *Lex*.
int Yyint Contains the integer value most recently parsed by *Lex*.

17 The Name Table

All the names used in the simulator (for units, unit types, sites, states, functions and sets) are stored in the global name table. User code may also insert names into the name table, look them up, and delete them. The name table access functions are:

```
char * EnterName ( name, type, data1, data2, data3 );
char * AlterName ( name, type, data1, data2, data3 );
NameDesc * FindName ( name, descriptor-pointer );
    char *name;
    int type, data1, data2, data3;
    NameDesc * descriptor-pointer;
```

EnterName will enter the *name* in the table. *type* is used by the simulator to determine what type of name it is. Type numbers 0 through 8 are used by the simulator, and numbers 9 through 99 are reserved to the simulator. Libraries should use numbers 100 through 999, and user code numbers 1000 and up. The three *data* fields are simply entered in the table. If the name is successfully entered, the function returns the pointer to the stored name character string. Care should be taken not corrupt this character string. *EnterName* will fail and return NULL if the name is already in the symbol table, unless all fields match those already in the table.

AlterName is just like *EnterName*, except that it never fails: if the name is already in the table, it is simply overwritten. This function should be used with extreme care.

FindName looks up the name in the name table, and fills in the descriptor with the table entry contents. It returns a pointer to the descriptor if the name was found, and NULL if not. The descriptor is of type *NameDesc* (see the Reference Manual for details), so a typical call to *FindName* might look like:

NameDesc descriptor;

```
if (FindName ( "some-name", &descriptor ) == NULL)
    printf("can't find name: %s\n", "some-name");
else
```

The *type*'s defined by the simulator are as follows:

Typevalue	used for
SCALAR	single unit name
VECTOR	unit vector name
ARRAY	unit array name
SET_SYM	set name
STATE_SYM	state name
STRING_SYM	unused name (may be re-used)
TYPE_SYM	unit type name
FUNC_SYM	function name
SITE_SYM	site name

Name table entries have the following structure:

```
typedef struct n_i_desc
{
    char    *name; /* Pointer to name */
    short   type;  /* Type of unit {0=SCALAR,1=VECTOR,2=ARRAY} */
    short   size;  /* Size of vector if VECTOR, number of columns if ARRAY */
    short   length; /* number of rows if type 2 */
    int     index; /* index of first unit name applies to */
    struct n_i_desc *next;
} NameDesc;
```

Each name must have a unique use, i.e. one cannot use the same name for a site that is used for a unit, type, function, state, set, etc. etc.

18 Customizing unit, site and link data structures

Each unit, site and link structure contains a field, *data*, which is for general purpose use. This field is the size of an integer or float, depending on which simulator is being used, but in any case is assumed to be the same size as a pointer. Therefore it is possible to use this field as a pointer to an arbitrary user-defined data structure. The simulator uses types *unit_data_type*, *site_data_type* and *link_data_type* for the unit, site and link *data* fields respectively. By using the *-D* flag in *makesim* (see section 23 and the man page for *makesim*) the user can re-define the type but must define it to be of size four bytes, so that user code is compatible with the simulator code.

18.1 Redefining the data type

Suppose, for instance, one wanted to delay the incoming values on a link by an arbitrary time steps. Then the *data* field for each link could be used as a pointer to a structure which stored the previous input values and weights. For example:

```
typedef struct inp
{
    short weight;
    short value;
} input;

typedef struct l_d_type
{
    short count;
    input * inputs;
} link_data;

typedef link_data * link_data_type;
typedef int site_data_type;
typedef int unit_data_type;
```

Here *unit_data_type* and *site_data_type* are defined to be *int*, just as the default. But *link_data_type* is now a pointer to a structure of type *link_data*, which contains a *count* field specifying the length of the propagation delay on the link. It also has a pointer, *inputs*, to a vector of type *input*, each element of which contains fields for *weight* and *value*.

18.2 Making a link

Since the simulator itself has no knowledge of this redefinition, it will not allocate space for the *link_data* structure, nor for the vector of *input*'s. Thus the user code must allocate the space, conventionally at the same time that *MakeLink* is called to make the link. For example, user code would do something like:

```
...
    lp = MakeLink(from,to,'excite',1000,0,LFdelay);
    MakeLinkData(lp,delay);
...
```

where *delay* is the number of simulation steps delay and *MakeLinkData* is:

```

MakeLinkData(lp,count)
    int count;
    Link * lp;

{
    int i;

    lp->data = (link_data_type) malloc (sizeof(struct l_d_type));
    lp->data->count = count;
    lp->data->inputs = (input *) malloc (sizeof(input) * count);
    for (i = 0; i < count; i++)
        lp->data->inputs[i].weight = lp->data->inputs[i].value = 0;
}

```

The incoming parameters are the pointer to the link, and the count of the number of steps delay. First the *link_data* structure is *malloc* ed, and the pointer to it stored in the link data *field*. Then the *count* is stored in the *malloc* ed structure, and the vector of length *count* and type *input* is *malloc* ed, the pointer to it being stored in the *link_data* structure. Finally all values and weights in the vector are initialised to zero.

18.3 The link function

Now the link function, *LFdelay*, would simply shift the values along one place, and store the current input value in the final vector location.

```

LFdelay(up,sp,lp)
    Unit * up;
    Site * sp;
    Link * lp;

{
    int i;

    for (i = 0; i < lp->data->count - 1; i++)
        lp->data->inputs[i] = lp->data->inputs[i+1];
    lp->data->inputs[i].weight = lp->weight;
    lp->data->inputs[i].value = *(lp->value);
}

```

The *for* loop shifts the vector entries up one place (towards the first entry - or top if it is thought of as a stack). Then the incoming values are stored in the final entry, or bottom of the stack.

18.4 The site function

Suppose we just wanted a standard weighted sum to be computed at the site, but using these delayed inputs. The function could be:

```
SFdelayweightedsum(up,sp)
    Unit * up;
    Site * sp;

{
    Link * lp;
    int sum = 0;

    for (lp = sp->inputs; lp != NULL; lp = lp->next)
        sum += lp->data->inputs[0].weight * lp->data->inputs[0].value;
    sp->value = sum/1000;
}
```

The for loop simply sums the weighted inputs from the *top* of the stack of delayed inputs, and scales the result appropriately.

18.5 Linking with simulator code

Since the unit, site and link structures are specified (at the C level) in a file that is included in user code, one might wonder how this code will ever compile. The solution, as hinted at above, is to provide a flag to the *makesim* command (see section 23) that specifies a file of user defined data structures to include instead of the standard ones. This is the -D flag, check the *makesim* man page. The file must define the types *unit_data_type*, *site_data_type*, and *link_data_type*. These types must be of the same size as an integer or float. Although the simulator code will continue to treat the *data* fields as an integer, since the simulator never actually uses these fields, it is permissible for user code to treat the field as a pointer, or any other integer-sized structure.

18.6 Displaying, saving, loading, etc

The problems arise in displaying, saving and loading. By default the simulator would display, save, and load the *data* field as an integer or float. The pointer value would not be restored on a load, and the extra structure would not be malloced. The solution to this problem is for the user to write specially named functions to display, save and load these structures. In addition the general help information provided by the simulator when the *Help* command (see section 20) is used might need to be augmented. The functions that may need to be written are as follows:

User_Unit_Display(fp,up) called when the *Unit* structure is displayed.

User_Site_Display(fp,up,sp) called when the *Site* structure is displayed.

User_Link_Display(fp,up,sp,lp) called when the *Link* structure is displayed.

User_Link_List(fp,up,sp,lp) called when links are listed.

User_Unit_Checkpoint(fp,up) called when the *Unit* structure is checkpointed.

User_Site_Checkpoint(fp,up,sp) called when the *Site* structure is checkpointed.

User_Link_Checkpoint(fp,up,sp,lp) called when the *Link* structure is checkpointed.

User_Unit_Restore(fp,up) called when the *Unit* structure is restored.

User_Site_Restore(fp,up,sp) called when the *Site* structure is restored.

User_Link_Restore(fp,up,sp,lp) called when the *Link* structure is restored.

User_Unit_Save(fp,up) called when the *Unit* structure is saved.

User_Site_Save(fp,up,sp) called when the *Site* structure is saved.

User_Link_Save(fp,up,sp,lp) called when the *Link* structure is saved.

User_Unit_Load(fp,up) called when the *Unit* structure is loaded.

User_Site_Load(fp,up,sp) called when the *Site* structure is loaded.

User_Link_Load(fp,up,sp,lp) called when the *Link* structure is loaded.

User_Help_Info() called after the general help information has been printed when the *Help* command is used.

where the arguments are a file pointer for the display information to be written to and pointers to the *Unit*, *Site* and *Link*.

If these functions exist in user code, then the simulator will call them to handle the *data* field for units, sites, and/or links, and to print extra general help information. If they do not exist, the default simulator action of treating the field as an integer or float will be taken. Examples of some of them follow, using the propagation delay definitions given in the preceding sections.

18.7 Displaying units

```
User_Link_Display(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;

{
    int i;

    for (i = 0; i < lp->data->count; i++)
        fprintf(fp, "\t\t%d delay - value: %d weight: %d\n", i+1,
            lp->data->inputs[i].value, lp->data->inputs[i].weight);
}
```

This function is called when the *display* command is issued from the simulator command interface. Instead of printing the link *data* field, the simulator calls this function to display the information. The display information is written to the file pointer, with *lp* being a pointer to the link currently being displayed, *sp* a pointer to the site at which the link arrives, and *up* a pointer to the unit to which the site is attached. In this case the latter two pointers are not used, but they are passed as parameters to the functions dealing with links for completeness. As can be seen, the function simply prints out the stack of delayed input values and weights. Then when a unit is *displayed*, the link would be printed in the following fashion:

```
link:**NO NAME** (3) func:LFdelay weight 1000 value:0
      1 delay - value: 0 weight: 0
      2 delay - value: 0 weight: 0
      3 delay - value: 0 weight: 0
```

18.8 Listing links

The *list* command lists each link, and as part of the display prints the value of the link *data* field. In our example we might wish to have the delay associated with the link displayed instead.

```
User_Link_List(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;

{
    fprintf(fp, "delay: %d\n", lp->data->count);
}
```

As in the link display function in the previous section, the file pointer and pointers to the unit, site and link are passed in as arguments. The function simply prints the delay value in the *count* field.

18.9 Checkpointing and Restoring

When a *checkpoint* command is issued from the simulator command interface, the state of the network is saved to file, that is the values in the data structure. On a *restore* command, these values are restored from the file. To ensure that the extra structures we have defined and created are also *checkpointed* and *restored*, the following functions are written:

```
User_Link_Checkpoint(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;

{
    int i,j,k;

    fprintf(fp, " %hd",lp->data->count);
    for (i = 0; i < lp->data->count; i++)
        fprintf(fp, " %hd %hd",lp->data->inputs[i].weight,
                lp->data->inputs[i].value);
    fprintf(fp, "\n");
}

User_Link_Restore(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;

{
    int i, count;

    fscanf(fp, "%d",&count);
    for (i = 0; i < count; i++)
        fscanf(fp, "%hd %hd",&(lp->data->inputs[i].weight),
                &(lp->data->inputs[i].value));
}
```

User_Link_Checkpoint prints out the *count* field, corresponding to both the delay on the link and the length of the vector of incoming weights and values. Then it prints out the vector of weights and values. *User_Link_Restore* simply does the complimentary thing: it reads in the count, and then reads in that number of incoming weight/value pairs, and saves them in the vector. The only important thing about these functions, and it is *crucially* important, is that the *Restore* function reads in exactly the same amount of data that the *Checkpoint* function wrote out. Otherwise, the *restore* process will fail.

18.10 Saving and Loading

When a *save* command is issued from the simulator command interface, the structure and state of the network is saved to file. On a *load* command, the network is built in the simulator, and the state reset. The saving and loading of the state are handled by an internal call to the *checkpoint* and *restore* process, so that the only extra thing that needs to be done is write functions to save the structure of our extended links, and recreate that structure when loading.

```
User_Link_Save(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;

{
    fprintf(fp, " %d ", lp->data->count);
}

User_Link_Load(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;

{
    int count;

    fscanf(fp, "%d", &count);
    MakeLinkData(lp, count);
}
```

The only important factor in the extended link structure is the length of the vector of delayed input weigh/value pairs. Thus *User_Link_Save* simply writes out this length. When loading, the extra structures we defined have to be explicitly created; the link is being made afresh. *User_Link_Load* reads in the size of the *vector*, and calls the data structure creation function *MakeLinkData* (see section 18.2) to allocate and initialise the space. The values will be restored by *User_Link_Restore* when the *restore* process is internally called by the simulator.

Once again, the most crucial thing to get right with these functions is that the *Load* function reads in exactly the same amount of data as the *Save* function writes out.

18.11 Unit and Site functions

In our example, the unit and site *data* fields are simply the simulator default integers. If they had been redefined to be some structure, we would need to write corresponding functions to deal with them, named as in section 18.6.

18.12 Completing the example

Using the functions described above, we may complete the example with a function to build a small network. This example is included in the */example/userdef* subdirectory. The build function simply creates ten units, each one linked to the other nine, with the delay on each link set to the absolute difference between the unit indices. This is not meant to represent anything, just to provide a simple example.

```
build()

{
  int i,j,k, n;
  Link * lp;

  AllocateUnits(10);
  for (i = 0; i < 10; i++)
  {
    u = MakeUnit('neuron',UFsum,0,0,0,0,0,0);
    AddSite(u,'excite',SFdelayweightedsum,0);
  }
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      if (i != j)
      {
        lp = MakeLink(i,j,'excite',1000,0,LFdelay);
        MakeLinkData(lp,(10+i-j)%10);
      }
}
```

The only departure from conventional network construction code is the call to *MakeLinkData* as each link is constructed.

19 Customizing the simulator command interface

Any user written function which has a name commencing "Cmd_" will be treated as a regular command by the simulator, and will be available at all interfaces. The simulator passes an *argc-argv* structure to command functions. There is a standard format for command functions, one aspect of which is obligatory. The function must allow for the case when *argc* is 2 and the *argv[1]* is the string "?". This occurs when the user types *command ?* or *help command*. In this case the function should print help information, including the command syntax. Conventionally single character command functions, e.g. *Cmd_e*, are abbreviations for other commands, and so are not listed when the user types ? to the prompt.

Suppose one wanted a command, *linkdata*, to set one of the delayed weight/value pairs in a link (as in the example above). In canonical form, this might look like.

```
#include "lex.h"

Cmd_linkdata(argc,argv)
    int argc;
    char ** argv;

{
    int from, to, delay, weight, value;
    char sitename[100];
    Unit * up;
    Site * sp;
    Link * lp;

    if (argc == 2 && Lex(argv[1]) == HELP)
        goto helpinfo;
    if (argc != 7) goto synerror;
    sscanf(argv[1], "%d", &from);
    sscanf(argv[2], "%d", &to);
    sscanf(argv[3], "%s", sitename);
    sscanf(argv[4], "%d", &delay);
    sscanf(argv[5], "%d", &weight);
    sscanf(argv[6], "%d", &value);
    up = &UnitList[to]; /* get unit pointer */
    for (sp = up->sites; /* find site */
         sp != NULL && strcmp(sp->name, sitename);
         sp = sp->next);
    if (sp == NULL) goto synerror;
    for (lp = sp->inputs; /* find link */
         lp != NULL && lp->from_unit != from;
         lp = lp->next);
    if (lp == NULL) goto synerror;
    if (delay > lp->data->count) goto synerror; /* check delay in range */
    lp->data->inputs[delay-1].weight = weight; /* set new values */
    lp->data->inputs[delay-1].value = value;
    return 0;

helpinfo:
    Format = TRUE; /* print detailed help */
    LOGfprintf(Disp, "The linkdata command is used to set the weight
and value in a time delayed link. You must give, in order, the unit
from which the link originates, the unit to which it goes, the site at
```

which it arrives, which delay you want to adjust the values for, the new weight, and the new value\n');
 Format = FALSE;

synerror:

```
LOGfprintf(Dispf,
  '\nUsage: linkdata [<From-UnitID> <To-UnitID> <To-site>
                    <delay> <weight> <value>]\n\n');
```

return 0;

}

Much more error checking should be done, but for brevity it is omitted here. The important point is the initial check for the *help* condition described above (note the use of *Lex*). If it exists, there is a jump to the help information label *helpinfo*. Here the simulator global variable *Format* is set to TRUE so that *LOGfprintf* will format the output (see section 4). The help information is written to *Dispf* since *help* is one of the commands whose output is piped. The string inside the call to *LOGfprintf* contains no newlines - for the purposes of this documentation it has been split into lines. After the help information has been displayed, formatting is switched off.

The other label, *synerror*, is conventionally used if a syntax error is discovered while processing the command *argc-argv* structure. The command syntax is printed, again to *Dispf* since conventionally the help information should include the syntax specification. If piping is turned off *Dispf* will be *stdout*, so everything works even if *synerror* is jumped to from elsewhere in the command function code.

Any function whose name begins with "Debug.Cmd_" will be available at the debug interface, but not the normal interface. It will take precedence over any normal command of the same name. For example, to create a different *linkdata* command at the debug interface, the function would be called "Debug.Cmd_linkdata". This function would be called when the *linkdata* command was issued at the debug and interrupt interfaces, but "Cmd_linkdata" would be called at the normal interface.

All command functions should return 0, whether there is an error or not, with the exception of any command used to exit the debug interface, such as the simulator function *Debug.Cmd_quit*. Any such function should return TRUE. *Errors* is a bit vector used within the debug facility. One cannot exit the debug facility to the normal interface until *Errors* is zero.

20 Adding to the Help information

If a simulator has been customized it may be that extra information describing the customization should be printed with the general help information (i.e the message that is printed when the command *help* is issued). The customizer can write a function, *User_Help_Info*, which will be called by the simulator after the standard help information has been displayed. Continuing the example in the preceding two sections, such a function could be:

```
User_Help_Info()
{
    Format = TRUE;
    LOGfprintf(Dispf, "This simulator is augmented to model fibre
propagation delays.  If you display a unit, you will notice each link
has a series of delays.  Every link has at least one delay.  At a
given time step, the current incoming value and the current weight are
saved at the bottom of the list.  At each time step the values and
weights percolate up the list one place.  When a value/weight pair
reaches the top of the list, it is used by the site function.\n\n");
    Format = FALSE;
}
```

As with the command function, formatting is switched on for the call to LOGfprintf, and off after it.

21 Avoiding name clashes

To avoid name clashes, do not use any names for functions, variables, data types, etc that begin with *si_*. These are reserved to the simulator code.

22 Floating Point version

The floating point version of the simulator uses floats for the *weight*, *data*, *output*, and *potential* fields of the unit, site and link structures. In addition the function pointers for the unit, site and link functions are declared *float ** instead of *int **. User code will be compiled with the -DFSIM flag so that conditional compilation is possible if one wants to freely move the code between integer and floating point simulators. In addition the type *FLINT* is defined to be an *int* in the integer simulator and a *float* in the floating point simulator.

23 Linking user and simulator code

User code must be compiled and linked with the simulator object files to form an executable simulator. *makesim* is a UNIX Bourne shell script to perform this somewhat complex process. The steps that must be taken are:

1. Compile the user source files.
2. Extract the names of user functions from the user object files and build a program to load the function names and pointers into the simulator function table. Thus the simulator will have a mapping from function name to function pointer, allowing the user to call the functions from the simulator command interface. This process also finds the function names in all libraries. The program *grabnames* interrogates object and library files and its output is fed to the program *makebind* which builds the appropriate C source file.
3. Compile the C source file created in the previous step.
4. Link the user object files, the object file created in the previous step, and the appropriate simulator object files.

makesim has a number of flags, which are described in the *man* page. Note that the user must have write permission in the simulator source directories to use the *-x* (compile simulator development version) flag. There is a mechanism in *makesim* for checking that user files have been compiled for at most one type of simulator (i.e. either integer or float). Briefly, the standard include file for user code (either *sim.h* or *fsim.h*) contains a specially named statically declared integer, with different names for the different types. *makesim* checks the user object files to ensure that at most one of these names exists.

The Rochester Connectionist Simulator
Volume 4:
Graphics Interface Programmer's Manual

Kenton Lynne
Dept. of Computer Science
University of Rochester
Rochester, NY 14627

May 11 1987

Contents

1	Introduction	2
2	Structural Organization	2
3	External Variables and Functions	2
4	Data Structures	3
5	Processing Strategy	4
6	Link Mode processing	5

1 Introduction

This document is directed to the programmer who needs to understand the working of the GI interface code. Although the code itself is well documented at the module level, this text is meant to provide a high level discussion of the underlying principles and organization of the software package. This document does NOT contain much user useful information: for information on how to USE GI, refer to the GI User's Manual and Guide which is contained in the file gi.doc.

2 Structural Organization

The GI code is distributed among more than a dozen source files. The primary organization revolves around the panel. There are six panels and thus six of the source files (obviously named) contain the code necessary to implement the the panel itself as well as most of the routines used to access the panel and its data as a unit. In addition there are several other source files whose organization revolves around function:

main.c contains the *mainline* (a misnomer), that is the entry point for setting up the GI tool to begin with. This file actually sets up the tool windows and then passes control to the Suntool *select* mechanism.

update.c contains the routines used to update the display after something has happened (such as a simulation step) and it is necessary to update the display panel with the new information for the displayed unit icons.

show.c contains the routines that perform the SHOW, CHANGE and ERASE functions that manipulate the display panel in terms of unit icons being displayed, changed or erased.

misc.c contains various miscellaneous routines used by several of the other modules. They have all been put in this one place for convenience since they don't really "belong" to any one particular function

gisim.c contains all the routines that access routines or data structures of the simulator itself. All interfaces to the simulator by GI routines should go through a routine in this file. We wish to segregate these simulator interface routines so that we can localize changes to the simulator interface (for example, the parallel version of the simulator on the Butterfly) to routines in this one file, rather than having to make changes (and have separate versions of the code) everywhere.

gistart.c contains the interface to the entry point. This routine and this routine alone is processed by the nametable function and the presence of the function name `gi_start` in the simulator's function table tells the simulator that gi has been compiled into the object code.

3 External Variables and Functions

GI uses many external (global) variables and functions. Most of these are defined in `globals.c` and externalized in `externs.h`. However some globals that are closely tied to certain routines are defined in those routines and externalized in separate header files for that file. For example, most of the Suntool Panel item variables that are externalized are defined in the corresponding `*.panel.c` file and externalized in `*.panel.h`.

Note that all external GI variables and functions begin with the suffix `gi_`. This is to enable the user, whose functions will be linked with the GI code, to avoid external name clashes. Any new GI external variables and functions should continue to follow this convention.

Most of the global defines and structure definitions are contained in the file `macros.h` which is an include in every source file. Anything added or changed in this file will be globally available to all the GI modules, but by the same token, will necessitate a complete recompile of all the GI sources.

4 Data Structures

The most complex data structures in GI are used to maintain the objects on the display panel. There are basically three kinds of graphic objects that can appear on the display:

grobj graphic objects for an icon that is tracking some aspect of a network unit. There is one of these for every icon on the display panel, maintained as a separate structure on one of two doubly linked lists.

txobj graphic object for a text item that appears on the display. There is a separate txobj structure for each line of text on the display and they are all linked together on their own doubly linked list.

drobj graphic objects for a drawn object consisting of 2-10 vertices joined by line segments. There is a separate drobj structure for every connected drawn object and they too are linked together on their own doubly linked list.

In order to optimize performance, two list of grobjs are always maintained. One contains all the grobjs that are currently displayed. The list header for this chain is called `gi_marker` because the marker icon itself resides inside the header structure. The other chain contains all the other grobjs that exist but are currently outside of the display panel window. The header for this structure is named `gi_off_display` and its structure is not used for anything currently. Any time the display panel changes size or the panel window is moved or a reshown with a new origin is done, these two chains have to be updated so that each contains the right grobjs for the new window. The routines that accomplish this are located in `update.c` source file.

Each of the grobj structure point to structures that maintain the information on the icon family that control the appearance of the icon relative to the aspect of the unit it is tracking. These icon structures (called gricons) contain pointers to array of `pixrect` pointer that make up the icon family as well as variables indicating how many members in the family and what the dimensions of the icons are. It also contains a pointer to the icon family name if this structure contains a user defined icon family. These gricons themselves are chained together on a singly linked list in order to be able to search for previously defined user icons.

All strings, vertices (of drawn objects) and fonts are maintained in separate areas using a similar strategy. For example string space is maintained in any number of separate buffers all of which are allocated dynamically as `STRING_SPACE_SIZE` bytes. Initially no string space is allocated. As soon as the first string needs to be stored a chunk is allocated and pointers maintained to the free area. When the chunk is used up, another buffer is allocated and so on. Since strings once stored always need to be retained these buffers are never deallocated until the GI exits. A similar strategy is followed for maintaining vertices and font definitions.

The only other data structure of consequence is the one (`info_unit`) that maintains the columns of information on the info panel. This is maintained as an array of 8 (`MAX_INFO_COLS`) structures which keep track of what unit is being tracked by the column as well as the current values for all aspects of the unit that appear in the column.

5 Processing Strategy

Most of the complexity of the code revolves around trying to be as smart as possible in NOT writing graphics object to the screen unless it is absolutely necessary. That is the reason two chains of grobjs are maintained - the objects on the off_display chain are not even looked at during the simulation. However even the grobjs on the on the displayed chain have a number of flags designed to increase the efficiency of the display actions. There is also one other important consideration in the update strategy. That is that the actions of getting the new values for the units after a simulation step is completely separate from the eventual display of the unit on the display panel. It is crucial to performance that these actions be strictly segregated for the version that will run on the butterfly. This is because all the values for all the units on the display are gotten at the SAME time (in one large buffer) in order to take advantage of the parallelism in the butterfly version of the simulator. Thus instead of looking at each unit on the chain, getting its value and then displaying it, getting the next unit's value, displaying it, etc. we choose to get ALL the unit values first and then display them all. Although this requires going through the display chain twice (once to get the values, then to display them), this is a small price to pay for the expected efficiencies to be gotten with the Butterfly.

The flags for controlling what needs to be gotten and/or reshown and what doesn't is contained in `gi_reshow_flag` which has a number of bit flags which need to be set or not depending on what kind of display behavior is called for. The `gi_reshow_flag` are set in various places of the code and checked and reset in the routine `gi_reshow`. The bit flags in `gi_reshow_flag` have the following meaning and should be strictly adhered to:

RESHOW_NEEDED indicates that there is something for `gi_reshow` to do (ie. there is at least one unit icon that may have to be updated). This flag is set on by any routine which through its processing knows that an object on one of the display chains may need to be reshown.

RESHOW_ALL indicates that all units on the display must be redisplayed regardless of the settings of their switches. This is set, for example, when the display window moves, necessitating the redisplay of everything on the display panel at different coordinates.

SOME_VALS_NEEDED indicates that some grobjs on the display chain need to have their values updated via the simulator. These grobjs are indicated by the fact that their `VALUE.OK` flags are off

ALL_VALS_NEEDED indicates that ALL grobjs on the display need to have their values updated regardless of the settings of their `VALUE.OK` switch. This flag would be set on when the simulator has run a simulation step, or indeed if any command is sent to the simulator.

SOME_LINKS_NEEDED analogous to **SOME_VALS_NEEDED** except it only applies in link mode to the Link value that the unit icon is displaying. It indicates that at least one grobj (with `LINK.OK` off) is on the display chain which needs to have its link value gotten from the simulator.

ALL_LINKS_NEEDED analogous to **ALL_VALS_NEEDED** except it only applies in link mode to the unit icon Link values. Indicates that ALL link values must be gotten from the simulator regardless of what the individual grobj flags indicate.

CLEAR_NEEDED indicates that `gi_reshow` should first clear the display panel before rewriting it. Caused when the display is moved to another part of display space or when the **RESHOW** button is pressed. Should thus take care of any "damage" on the screen.

What values have to be gotten and when a display of an icon is necessary is determined by the bit flags in each gricon. The critical flags and what they mean are as follows:

DISPLAYED on - indicates that the unit is currently displayed, and thus does not need to be redisplayed unless the display moves or its value changes

VALUE_OK on - indicates that the value in the *val* field is current and thus there is no need to go to the simulator. (Unless **ALL_VALUES_NEEDED** flag is on).

VALUE_CHG on - indicates that the value currently in the *val* field is correct but has been changed from the last time the unit has been displayed. Thus suggests that a new icon index and thus a different icon may be necessary to reflect the value of the unit on the display screen.

LINK_OK analogous to **VALUE_OK** except only applies in link mode to the *link_val* field.

LINK_CHG analogous to **VALUE_CHG** except only applies in link mode to the *link_val* and *link_index* fields.

6 Link Mode processing

Each grobj maintains two independent *values*: one for the current aspect of the unit it is tracking, and one maintained explicitly for link mode representing a link weight between it and the current *target* unit/site. Since there is only one target for all units in link mode, the target unit and site are maintained in global variables *gi_cur_link_target*, *gi_cur_link_site* and *gi_cur_link_ptr*. However if a unit is specifically tracking a particular aspect outside of link mode then the target information is maintained locally within the grobj itself.

Currently there is no support for "commands" for displaying links in link mode, mostly because it is felt that link mode is primarily an interactive task and doesn't have much value in command mode. However this could change in the future.

The Rochester Connectionist Simulator
Volume 5:
Back Propagation Library User Manual

Nigel Goddard and Toby Mintz
Dept. of Computer Science
University of Rochester
Rochester, NY 14627

April 28 1987

Contents

1	Introduction	2
2	Building a back-propagation module	2
2.1	BPmodule	3
2.2	BPinput	3
2.3	BPhidden	3
2.4	BPoutput	4
2.5	BPteach	4
2.6	BPfire	4
2.7	BPlink	5
2.8	BPEndmod	5
3	Simulating the module	6
3.1	BPsetinput	6
3.2	BPsetteach	6
3.3	BPcycle	6
4	Example: learning 8-3-8 encoding	7
4.1	Constructing the network	7
4.2	Running a simulation	9
5	Activation and error-propagation functions	10
5.1	Writing error propagation functions	10
6	Module unit layout and operation	11

1 Introduction

A simple back propagation network consists of a number of *layers* of units. The first layer is the *input* layer, the final layer the *output* layer, and the remaining intervening layers are *hidden* layers. Links within the network are feedforward: no links within a layer or from a higher layer to a lower layer are allowed.

A back propagation network has two modes of operation. When operating in the *forward* mode, activation spreads from lower layers to higher layers, using the unit *activation* functions. When operating in the *error-propagation* mode, errors are propagated from higher layers to lower layers, using an *error-propagation* function, units adjusting the weights on their incoming links as the errors propagate. This package allows arbitrary *activation* and *error-propagation* functions. The standard UCSD sigmoid activation function and derivative based error-propagation functions (see [1]) are provided in the package, but other functions may be written by the user.

Although many researchers have concentrated on looking at a single multi-layer back-propagation network, in general one would like to be able to include a back-propagation network (or several such networks) as sub-parts of a larger network. This package is designed to allow one or more back-propagation *modules* as part or all of a network.

A *cycle* is defined to be the process of feeding activation forward until the output layer has settled, then propagating errors back to the first hidden layer. The number of simulation steps this will take is twice the number of layers (only *hidden* and *output* layers, explained later) in the back-propagation module (the simulator *must* be running in synchronous mode).

2 Building a back-propagation module

The standard backpropagation module has certain required features, and some optional features. The required features are the following: a *control* unit to monitor and control the running of the module, a *bias* unit that can be optionally linked to units in the module, and an *output* layer of units. In almost all cases there will be at least one *hidden* layer of units, but it is not necessary. The optional features are as follows: an *input* layer of units which feeds information to the first *hidden* layer, and a *teach* layer of units which always has the same number of units as the *output* layer. It feeds information into the *output* layer, in effect telling it what the correct output is. Finally there is an optional *fire* unit which is used to trigger the *control* unit to tell it to start the module.

A back-propagation module is built with a series of calls to library functions. First the module name is declared, then the *hidden* and *output* layers are specified. Finally the end of the module is indicated. Optionally *input* and *teach* layers may be declared, the input layer before any hidden layers, the teach layer after all the hidden layers. The functions to do this (in the order in which they should be called) are:

BPmodule declares the beginning of a module.

BPinput creates the optional *input* layer.

BPhidden creates a *hidden* layer.

BPoutput creates the *output* layer.

BPteach creates the *teach* layer.

BPlink creates a link between two units in the module.

BPfire creates the *fire* unit and links it to the *\$fire* site of the *control* unit.

BPendmod indicates the end of the module specification.

Once the module has been constructed, the trial patterns and correct outputs must be presented to the network during simulation. If the module is part of a larger network, other units in the network may feed the trial and output patterns to the module. Alternatively, functions are provided to set the trial and output patterns in the input and teach layers from a user program. These functions are:

BPsetinput sets up a trial pattern in the *input* layer.

BPsetteach sets up the correct pattern in the *teach* layer.

BPcycle turns the module on for a number of *cycles*.

2.1 BPmodule

The first function that must be called is *BPmodule*. It takes as arguments a string which is the name of the module, and an integer which is the number of layers (including only the *hidden* and *output* layers, not *input* or *teach* layers). This will set up the building process and create a *control* unit and a *bias* unit. If the name of the module is *foo* the control unit will be named "cont_foo" and the bias unit "bias_foo". For example:

```
BPmodule("learn", 3)
```

declares the beginning of a section of code that defines the module *learn* that will have two *hidden* layers and one *output* layer. The units "cont_learn" and "bias_learn" will be created. From now on, until the module is completed with the *BPendmod* routine, there should be no calls to the *MakeUnit* unit construction function from user code.

2.2 BPinput

After the module is started with the *BPmodule* command an optional *input* layer may be created. The routine *BPinput* takes as an argument one integer representing the number of units in the layer. It creates a unit vector of that length with name "*foo*(0)" (where *foo* is the module name). The units are created with the null function. For example:

```
BPinput(7)
```

following the above call to *BPmodule* would create a vector of seven units with the name "learn(0)" (not to be confused with "learn[0]"). Thus the third input unit will be called "learn(0)[2]". The outputs of the units in the *input* layer can be set at any time during simulation using the *BPsetinput* routine described in section 3.1. *BPinput* returns the index to the vector.

2.3 BPhidden

Now the *hidden* layers are created, each layer being a vector of units. They will be given names "*foo*(1)" to "*foo*(*n*-1)" where *n* is the number of layers specified in the call to *BPmodule* (if *n* is 1 then there are no *hidden* layers). The units in these layers have one site called the *\$learn* site. The routine *BPhidden* must be called for each *hidden* layer. It takes 11 arguments, the first is an integer representing the number of units in the layer. The second is a pointer to a unit function, the

standard function is supplied in the library and is called *UFh.o* (it is the same as the standard output layer function). The third argument is a pointer to a site function for the *\$learn* site, the standard function is *SFbbsigmoid*. The fourth argument is either 0 or 1 signifying if the *bias* unit is to be linked to the units in this layer. The fifth is the weight of the link from the *bias* unit; it must be supplied even if the bias unit will not be linked. The next six arguments specify what certain fields in each unit in the layer should be initially set to. They are all integers (in the floating point version only the last two are integers, the others are floats): initial potential, potential, data, output, initial state, and state. *BPhidden* returns the index to the vector. For example:

```
BPhidden(3, UFh.o, SFbbsigmoid, 1, 500, 0, 0, 0, 0, 0, 0)
```

creates a layer with three units. The units' unit function is *UFh.o* and their *\$learn* site function is *SFbbsigmoid*. The *bias* unit will be linked to each unit in this layer with a weight of 500. The remaining field values for the units are initialized to 0.

2.4 BPoutput

After ALL the *hidden* layers are created the *output* layer must be created. The units in the *output* layer are identical to those of the *hidden* layers except that there is one extra site, the *\$error* site. As a result the routine *BPoutput* takes identical arguments except there is one more which appears right after the *\$learn* site function pointer, thus moving the remaining arguments one space to the right. This is the *\$error* site function pointer; the standard function given in the library is *SFerror* (see 5.1). *BPoutput* creates a unit vector with name "*foo(n)*" where *n* is as above, and returns the index to the vector. For example:

```
BPoutput(5, UFh.o, SFbbsigmoid, SFerror, 0, 0, 0, 0, 0, 0, 0, 0)
```

creates a vector with five units. The units' unit function is *UFh.o* and their *\$learn* site function is *SFbbsigmoid*. The *bias* unit will not be linked to the units in this layer, and the field values for these units will be initialized to 0. If this call to *BPoutput* occurred after the above call to *BPhidden* then the layer would have the name "*learn(3)*."

2.5 BPteach

After the *output* layer an optional *teach* layer can be created. The routine to do this is *BPteach*. It takes no arguments and creates a vector of the same length as the *output* layer with name "*foo(n-1)*." Links are made from each unit in the *teach* layer to the *\$error* site of its corresponding unit in the *output* layer. The units have no function (null function pointer) and their outputs can be set using the *BPsetteach* routine described later. *BPteach* returns the index of the vector.

So calling *BPteach* after the above call to *BPoutput* would create a vector with five units and name "*learn(4)*."

2.6 BPfire

After the *output* layer (and the *teach* layer if there is one) there can be an optional *fire* unit. The routine that creates this unit is *BPfire*; it takes no arguments. It creates a unit with name "*fire.foo*" and links its output to the *\$fire* site of the *control* unit. It returns the index of the *fire* unit.

So calling *BPfire* after the above call to *BPteach* or *BPoutput* would create a unit called "*fire.learn*." The unit would be linked to the *\$fire* site of the unit "*cont.learn*." The unit function for the *fire* unit is automatically set to be *UFfire*, defined in the library.

Initially the output of the *fire* unit is 0. When its output is non-zero it triggers the *control* unit to start a *cycle*. Each time the *fire* unit runs it reduces its output by one. Therefore if a *cycle* takes 6 steps then running the network with the *fire* unit outputting 12 at the start would let the module run through 2 cycles.

The output of the *fire* unit can be set by the user with *BPcycle* described below. If there is no *fire* unit then the network will have to activate the *control* unit on its own.

2.7 BPlink

To link units in layers the routine *BPlink* is called: it takes 5 arguments. The first argument is the layer number of the unit where the link is coming *from*. It can be from 0 (*input* layer) to *n*-1 (last *hidden* layer). The second argument is the local unit index within the *from* layer which specifies the actual *from* unit. The third argument is the layer number of the unit that the link is going *to*. It can be from 1 (first *hidden* layer) to *n* (*output* layer). The fourth argument is the local unit index within the *to* layer which specifies the actual *to* unit. The fifth argument is an integer which is the weight of the link (float if floating point version). For example:

```
BPlink(0, 2, 1, 0, 250)
```

links the third unit of the *input* layer to the first unit of the first *hidden* layer, with a weight of 250.

Links can only be made in the forward direction, and no links can be made to the *teach* layer.

NOTE: Because the error-propagation function uses the link function pointer field as a backpointer with which to propagate errors there can *not* be any link function.

2.8 BPendmod

To complete the module the routine *BPendmod* is called with an argument which is the name of the module (given to *BPmodule*). After this is called only calls to the simulation functions *BPsetteach*, *BPsetinput* or *BPcycle* (see section 3) can be made (or *BPmodule* to start a new module). For example, to complete the module started by the *BPmodule* above we would call:

```
BPendmod("learn")
```

This would label the "learn" module as complete. An error will occur if the correct number of layers have not been created.

3 Simulating the module

During simulation the trial patterns and correct output patterns must be provided to the *input* and *output* layers. This can be done by other units if the module is embedded in a larger network, or by explicit library function calls from user code.

3.1 BPsetinput

To set the output values of input units the routine *BPsetinput* is called. It takes three arguments. The first is a string which is the name of the module. The second is an integer which is the local index of the unit within the input layer vector (starting at 0). The third is an integer (or float in floating point version) which is the value to set the output of the specified unit to. This routine can be called while the module is being built (after the *input* layer has been built) or after the module has been completed. For example:

```
BPsetinput("learn", 6, 900)
```

called anytime after the *input* layer has been created, will set the output of the seventh unit in the *input* layer to 900. An error is signalled if the unit index is out of range.

3.2 BPsetteach

To set the output values of *teach* layer units the routine *BPsetteach* is called. Its arguments are identical to those of *BPsetinput*. It can be called anytime after the *teach* layer is created. For example:

```
BPsetteach("learn", 4, 0)
```

will set the output of the fifth unit of the *teach* layer to 0. An error is signalled if the unit index is out of range.

3.3 BPcycle

BPcycle is called to activate the network for a number of *cycles*. This can also be done by feeding activation from another unit to the *\$fire* site on the *control* unit (whenever the *control* unit gets activation on that site it starts a *cycle*). *BPcycle* takes an integer parameter, the number of cycles to be run, and a string, the name of the module to be activated. For example:

```
BPcycle("concept-learn", 100)
```

will activate the module *concept-learn* for 100 cycles. Note that activating is *not* the same as simulation. In addition to the call to *BPcycle*, the simulator must be run for the appropriate number of time steps, i.e at least $\#cycles * \#steps-per-cycle$.

4 Example: learning 8-3-8 encoding

The goal behind the 8-3-8 encoding problem is to teach a network to encode a number between 0 and 7 into a '3-bit' code, and then decode it back into the original number between 0 and 7. The original number is represented as the activation of one of a group of 8 units. Which number is represented depends on which unit is activated. In other words, there is a unit for 0, a unit for 1, etc., up to 7. The '3-bit' representation consists of three units. When the network learns the encoding there will be a unique pattern of activation over these three units when given the activation of each of the original 8 units. The decoded number is represented in the same way as the original number.

4.1 Constructing the network

We will create a module with one *hidden* layer, one *output* layer, an *input* layer, a *teach* layer, and a *fire* unit. The *input* layer will consist of the 8 units which represent the original number. The *hidden* layer will consist of the 3 units which represent the '3-bit' code. The *output* layer will have 8 units like the *input* layer to represent the decoded number. The *teach* layer will be used to feed the correct result to the *output* layer. The standard functions will be used for both the *hidden* and the *output* layers.

Links will be made from each of the units in the *input* layer to each of the units in the *hidden* layer. Then from each of the units in the *hidden* layer to each of the units in the *output* layer. First the initial definitions are given. Both *sim.h* and *bp.h* must be included.

```
#include "sim.h"
#ifdef FSIM                                /* floating point simulator */
#   define BP_ONE      1.0
#   define BP_ZERO     0.0
#   define CAST        float
#else                                       /* integer simulator */
#   define BP_ONE      1000
#   define BP_ZERO     0
#   define CAST        int
#endif

#include "bp.h"
```

The above code can be compiled for either the floating point or integer version of the simulator. When the floating point version is made it is compiled *-DFSIM* so *FSIM* will be defined and *BP_ONE* will be 1.0 (rather than 1000), and *BP_ZERO* will be 0.0 (rather than 0).

```

build(argc, argv)
int    argc;
char   *argv[];
{
    register int    i,j;

    srandom(getpid());                /* seed random number generator */
    AllocateUnits(40);                /* upper limit on units */
    BPmodule(argv[1], 2);             /* start 2 layer module */
    BPinput(8);                       /* create input layer with 8 units */
    BPhidden(3, UFh_o, SFbpsigmoid, 1, /* create hidden layer */
             BP_ONE/2, BP_ZERO, BP_ZERO, BP_ZERO, BP_ZERO, 0, 0);
    BPoutput(8, UFh_o, SFbpsigmoid, SFerror, 1, /* create output layer */
             BP_ONE/2, BP_ZERO, BP_ZERO, BP_ZERO, BP_ZERO, 0, 0);
    BPteach();                        /* create teach layer */
    BPfire();                        /* create fire unit */
    for(i = 0; i < 8; i++)            /* create all links
        for(j = 0; j < 3; j++)
        {
            BPlink(0, i, 1, j, (random()%700)*(BP_ONE/1000)+(BP_ONE/10));
            BPlink(1, j, 2, i, (random()%700)*(BP_ONE/1000)+(BP_ONE/10));
        }
        /* weights random .1 - .8 */
    BPendmod(argv[1]);
}

```

The call to *AllocateUnits* is standard for building any network. The number of units allocated must include the units in the *input* layer (if there is one), in all *hidden* layers, in the *output* layer, and in the *teach* layer (if there is one). It also must include the *bias* unit, the *control* unit, and the *fire* unit (if there is one).

The call to *BPmodule* names the module being built and defines it as having 2 layers (since one layer must be an *output* layer that means there is only one *hidden* layer).

The call to *BPinput* creates the optional *input* layer with 8 units.

Then the *hidden* layer is created. It has 3 units, each unit having the function *UFh_o*, which is the standard unit function. The site function for these units is *SFbpsigmoid*, again the standard function. The '1' in the fourth argument position signifies that the *bias* unit is to be linked to each unit in this layer. The weight of the link is *BP_ONE/2*, which is 500 for the integer version, and .5 for the floating point version. The remaining 6 arguments set the initial potential, potential, data, output, initial state, and state values of each unit in this layer to 0. (The last two arguments are integers in either version of the simulator.)

The *output* layer is created with 8 units. It is created like the *hidden* layer except it has an extra site function, *SFerror*, for the *\$error* site which *hidden* layer units do not have.

The call to *BPteach* sets up a *teach* layer with 8 units (because the *output* layer has 8 units), and links the output of each unit in the *teach* layer to the *\$error* site of the respective units in the *output* layer.

Then linking is done, in the manner described above, between the *input* layer and the *hidden* layer, and then to the *output* layer. The weight for each link is a random number between 100 and 799 for the integer version, and between .1 and .799 for the floating point version.

The end of the module is signified by the call to *BPendmod*.

4.2 Running a simulation

To run a simulation, we shall use a function that sets up trial patterns and correct outputs in the *input* and *teach* layers. For example, unit 0 of the *input* layer is activated, as well as unit 0 of the *teach* layer; then the network is run through one *cycle*. The function below does this for all eight *input* and *teach* units. This function can be called from the simulator command interface.

```
cycle(argc, argv)
int    argc;
char   *argv[];
{
    register int    i,j,k;

    if(argc != 3)
    {   fprintf(stderr, "Wrong # args\n");
        return;
    }
    j = atoi(argv[2]);
    for(k = 0; k < j; k++)
        for(i = 0; i < 8; i++)
        {   BPsetinput(argv[1], i, BP_ONE);
            BPsetteach(argv[1], i, BP_ONE);
            BPcycle(argv[1], 1);
            Step(4);                      /* 2 layers times 2 */
            BPsetinput(argv[1], i, BP_ZERO);
            BPsetteach(argv[1], i, BP_ZERO);
        }
}
```

The first argument to the above function is the name of the module and the second is the number of times each of the eight units should be activated. Originally all the units in the *input* and *teach* layers are outputting 0. The function goes through each unit in those layers, one by one, and sets their outputs to 1000 (or 1) using *BPsetinput* and *BPsetteach*. The call to *BPcycle* sets up the module to be run for one *cycle*. Then the network is actually run, using the *Step* function. Then the units are reset to zero. This process is repeated as many times as the user specified in the function call.

The argument to *Step* is '4' because the number of steps in a *cycle* is 2 times the number of layers (*hidden* and *output* only). This example is contained in the *example/backprop* subdirectory.

5 Activation and error-propagation functions

There is a standard format for the unit functions to be used in the *hidden* and *output* layers. It is of the form:

```
In forward state (activation):
    Output some function of activation from $learn site (usually identity);
    call BPendfwd passing the unit pointer as argument.
In reverse state (error-propagation):
    For each link into $learn site:
        Change weight of link.
        Propagate error to unit at other end of link.
    call BPendrev passing the unit pointer as argument.
```

The library provides the standard UCSD functions for activation and error-propagation. Other functions may be written by the user.

5.1 Writing error propagation functions

Errors are propagated through links between units by the *link function pointer*. When the links are made with *BPLink* the link function pointer is set to point to the *data* field of the *\$learn* site where the link is originating from. Suppose unit *A* is linked to unit *B* via link *l*. When *B* propagates its error down to *A* it puts the error where *l*'s link function pointer is pointing; this will be to the *data* field of *A*'s *\$learn* site. Any user-written error-propagation functions should use this method. As an example consider the code for the standard error-propagation function, in the library.

```
FLINT
UFh_o(up)
Unit    *up;
{
    FLINT      delta,
              deltaw;
    Link      *lp;

    /*-----**
    ** activation code **
    **-----**/
    if(TestFlagP(up, FORWARD_FLAG))
    {   up->potential = up->output = up->sites->value;
        BPendfwd(up);
    } else
    /*-----**
    ** error-propagation code **
    **-----**/
    {
        delta = (up->sites->data * (up->output) * (BP_ONE - up->output))/(BP_ONE * BP_ONE);
        for(lp = up->sites->inputs; lp != NULL; lp = lp->next)
        {   deltaw = ((delta * *(lp->value))/BP_ONE)
            + BPmomentum * lp->data;
```

```

        lp->weight += deltaw;
        lp->data = deltaw;
        *(FLINT *) (lp->link_f) += (delta * lp->weight)/BP_ONE;
    }
    BPendrev(up);
}
}

```

First note that the symbol *FLINT* is defined to be *float* if compiled for the floating point version, and *int* if compiled for the integer version of the simulator. In addition, remember that *BP_ONE* is defined as 1.0 for the floating point version and 1000 for the integer version.

In the first part of the error-propagation section *delta*, the error to be propagated is computed. It is a function of the error passed down from an upper layer, stored in *up->sites->data*, and the current output of the unit, *up->output*. The specific function used is described in [1].

For each link coming into the *\$learn* site of the unit the weight change for each link, *deltaw*, is computed. It is a function of *delta* and the output of the unit where the link originates. In addition a *momentum* factor can be added in. The global variable *BPmomentum* is a floating point between 0 and 1. *lp->data* is set to the last weight change of the link (initially 0).

After the link weight change is computed the change is made to the current weight, and then it is stored in *lp->data* to be used to compute the momentum factor during the next error-propagation stage.

Finally the error *delta* is passed down to the unit at the other end of the link as a function of the weight of the link. In other words the propagated error is in proportion to the significance of the link. Note that the address *lp->link_f* must be type-cast as a float pointer or an int pointer (depending on the version of the simulator) so that the compiler will not interpret it as a function pointer. Also note that the propagated error is added: this is because a single unit might output to a number of units, and error should be totalled from all these units.

The library function *BPendfwd* switches the state of the unit from forward to reverse and sets the *NO_UNIT_FUNC_FLAG*. The library function *BPendrev* switches the state of the unit from reverse to forward, sets the *NO_UNIT_FUNC_FLAG*, and clears the accumulated error.

Errors originate at the *\$error* sites of *output* units. The *\$error* site function calculates the error of the *output* unit by comparing its output with data coming from the *teach* layer. It puts the error in the *data* field of the *\$learn* site of the *output* unit, thus allowing for standard error-propagation procedures. The library's *\$error* site function is *SError*.

6 Module unit layout and operation

Back-propagation module units appear in the *UnitList* in a particular order. The *control* unit comes first, then the *bias* unit, followed by the units in the *input* layer (if there is one). Next come the units of all of the *hidden* layers, the *output* layer, and the *teach* layer (if there is one). The last unit is the *fire* unit (if there is one).

If the name of the module is *learn* then the *control* unit would be named *cont.learn*, the *bias* unit *bias.learn*, and the *fire* unit *fire.learn*. Each layer would be named *learn(n)*, where *n* specifies the layer number of the particular layer. The *input* layer number is 0, *hidden* layers start at 1. The indices of these units may be found using the *NameToInd* function. Each layer is a vector, while the control, bias and fire units are scalars.

Initially all layers units have the *NO_UNIT_FUNC_FLAG* set, as well as the *BP_FORWARD_FLAG*. To start simulation the *control* unit turns on the first *hidden* layer of units by *unsetting* the

NO_UNIT_FUNC_FLAG. After these units run they reset their **NO_UNIT_FUNC_FLAG**'s. In addition, if the **BP_FORWARD_FLAG** is set it is then *unset* so that the next time the unit is turned on it will be in the *reverse* direction (if the flag was *unset* then it is *set* again). Then the *control* unit turns on the next layer of units. The process continues until the *output* layer has run. At this point the *output* layer is run again (this time in the reverse, error-propagating direction) and the *control* unit then turns on layers in the reverse order. When it reaches the first *hidden* layer the whole process is ready to start again.

NOTE: The unit flag **BP_FORWARD_FLAG** is defined as 12. Care should be taken that there is no conflict with flags defined in other packages.

References

- [1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Exploring the Microstructure of Cognition Volume 1: Foundations*, Bradford Books/MIT Press, Cambridge, MA, 1986.

END

DATE

9-88

DTIC